# CAN driver user guide

**TRAVEO™ T2G family**

## About this document

### Scope and purpose

This guide describes the architecture, configuration, and use of the controller area network (CAN) driver. This document explains the functionality of the driver and provides a reference of the driver's API.

The installation, build process, and general information on the use of the EB tresos Studio are not within the scope of this document. See the EB tresos Studio for ACG8 user's guide [10] for a detailed discussion of these topics.

### Intended audience

This document is intended for anyone who uses the CAN driver of the TRAVEO™ T2G family.

### Document structure

Chapter 1 General overview gives a brief introduction to the CAN driver, explains the embedding in the AUTOSAR environment, and describes the supported hardware and development environment.

Chapter 2 Using the CAN driver details the steps on how to use the CAN driver in your application.

Chapter 3 Structure and dependencies describes the file structure and the dependencies for the CAN driver.

Chapter 4 EB tresos Studio configuration interface describes the driver's configuration with the EB tresos Studio.

Chapter 5 Functional description gives a functional description of all services offered by the CAN driver.

Chapter 6 Hardware resources gives a description of all hardware resources used.

The Appendix A and Appendix B provides a complete API reference and access register table.

### Abbreviations and definitions

**Table 1 Abbreviations**

| Abbreviations | Description |
|---|---|
| API | Application Programming Interface |
| ASCII | American Standard Code for Information Interchange |
| ASIL | Automotive Safety Integrity Level |
| AUTOSAR | Automotive Open System Architecture |
| BSW | Basic Software. Standardized part of software which does not fulfill a vehicle functional job. |
| CAN | Controller Area Network |
| CAN-FD | Controller Area Network with Flexible Data rate |
| COM | Communication |

| Abbreviations | Description |
|---|---|
| | |
| DEM | Diagnostic Event Manager |
| DET | Default Error Tracer |
| DLC | Data Length Code |
| EB tresos Studio | Elektrobit Automotive configuration framework |
| ECU | Electronic Control Unit |
| EcuM | ECU State Manager |
| FIFO | First In, First Out |
| GCE | Generic Configuration Editor |
| HOH | Hardware Object Handle |
| HRH | Hardware Receive Handle |
| HTH | Hardware Transmit Handle |
| HW | Hardware |
| ICOM | Intelligent Communication Controller |
| IRQ | Interrupt Request |
| ISO | International Organization for Standardization |
| ISR | Interrupt Service Routine |
| µC | Microcontroller |
| MCAL | Microcontroller Abstraction Layer |
| MCU | Micro Controller Unit |
| OS | Operating System |
| PDU | Protocol Data Unit |
| RX | Received eXcgange |
| SW | Software |
| TX | Transmit eXchange |
| UTF-8 | 8-Bit Universal Character Set Transformation Format |

**Related documents**

**AUTOSAR requirements and specifications**

**Bibliography**

[1] General specification on basic software modules, release 4.2.2.

[2] Specification of standard types, release 4.2.2.

[3] Specification of ECU state manager, release 4.2.2.

[4] Specification of default error tracer, release 4.2.2.

[5] Specification of CAN driver, version 4.0.0, release 4.0, revision 3.

[6] Specification of CAN interface, version 5.0.0, release 4.0, revision 3.

[7] Specification of CAN driver, release 4.2.2.

[8] Specification of CAN interface, release 4.2.2.

[9] Specification of module PORT driver, release 4.2.2.

### Elektrobit automotive documentation

### Bibliography

[10] EB tresos Studio for ACG8 user’s guide.

### Hardware documentation

The hardware documents are listed in the delivery notes.

### Related standards and norms

### Bibliography

[11] Layered software architecture, AUTOSAR release 4.2.2.

# Table of contents

Table of contents

# 1 General overview

## 1.1 Introduction to CAN driver

The CAN driver abstracts the hardware CAN controllers of the TRAVEO™ T2G family microcontrollers and provides the API functions for sending and receiving messages. In addition, the CAN driver can trigger an upper layer such as the CAN interface by callback functions to indicate that new messages were received. The CAN driver can operate in interrupt driven or in polling mode.

Some characteristic properties of the CAN device are:

- Classic CAN and CAN-FD (ISO 11898-1:2015)
- Transmission/Reception of standard and extended frames
- Bit rate: up to 2 Mbps nominal, up to 16 Mbps data
- Up to 224 message buffers for each instance (sharing multiple controllers).
- Hardware RX message FIFO support / TX message queue support (multiplexed transmission)
- Support of classic CAN and CAN-FD mixed mode
- Pretended networking mode support (ICOM software implementation)

## 1.2 User profile

This guide is intended for users with a basic knowledge of the following:

- CAN
- Embedded systems
- AUTOSAR communication terminology
- C programming language
- Target hardware architecture

## 1.3 Embedding in the AUTOSAR environment

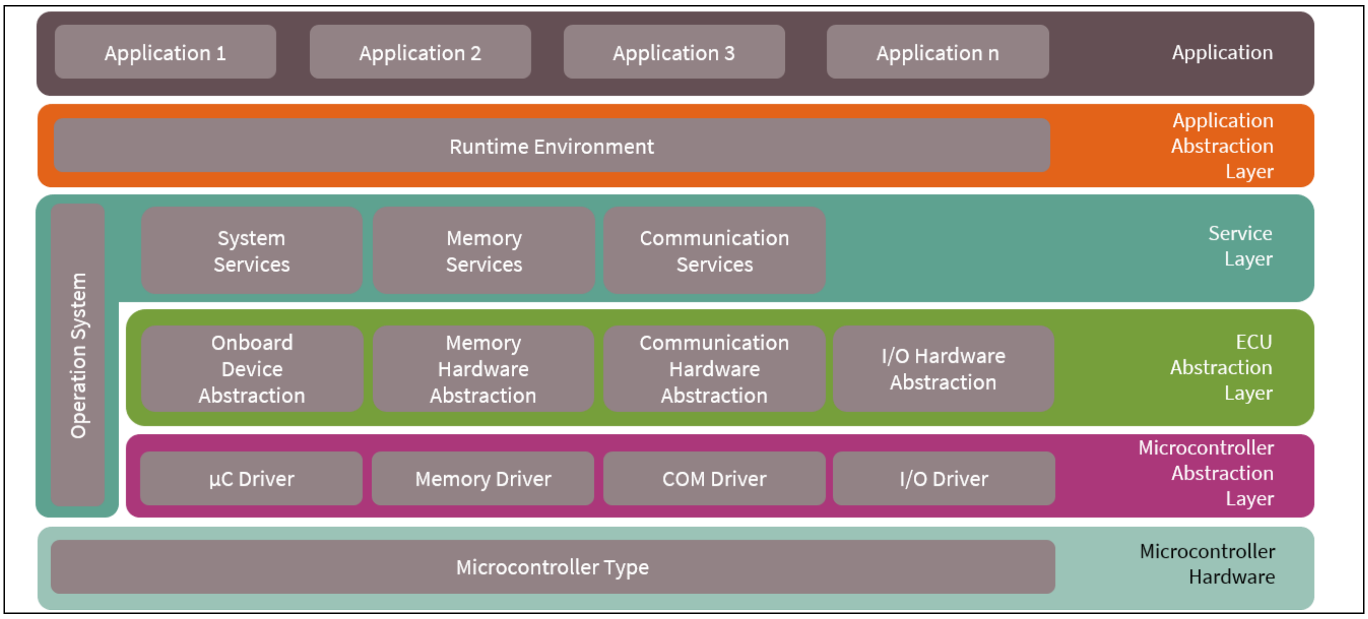

**Figure 1 Overview of AUTOSAR software layers**

Figure 1 depicts the layered AUTOSAR software architecture. The CAN driver (Figure 2) is one of the communication drivers in the microcontroller abstraction layer (MCAL), the lowest layer of basic software in the AUTOSAR environment.

As a communication driver, CAN driver accesses the hardware directly and provides a standardized and hardware independent API for the CAN interface.

For an overview of the AUTOSAR layered software architecture, see *Layered software architecture* [11].

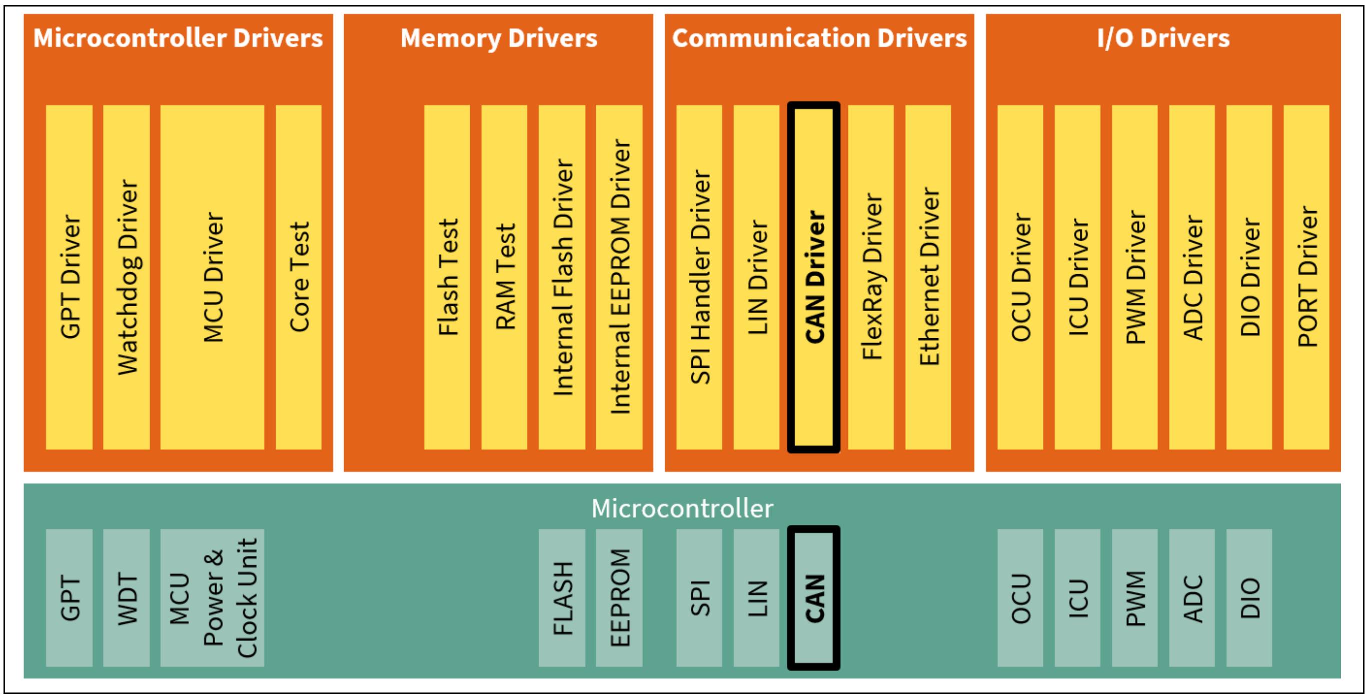

**Figure 2 CAN driver in MCAL layer**

## 1.4 Supported hardware

This version of the CAN driver supports the internal CAN controller of TRAVEO™ T2G family microcontrollers. External hardware devices are not supported.

The supported derivatives are listed in the release notes.

Smaller derivatives have only a subset of the ports and pins defined for the microcontroller. New derivatives will be supported on request or via resource file update. For an overview of all supported derivatives have a look at the Resource plugin.

Note: *External transceiver hardware and related driver software (CanTrcv) are required to communicate via a CAN bus network. They are not part of the TRAVEO™ T2G hardware or related MCAL software product.*

## 1.5 Development environment

The development environment corresponds to AUTOSAR release 4.2.2. The Base, Make, Mcu, Port, and Resource modules are required for the proper functionality of the CAN driver.

According to AUTOSAR release 4.2.2., the AUTOSAR environment must provide the *Can_GeneralTypes.h* file for proper CAN functionality. Please properly set include path for *Can_GeneralTypes.h*.

## 1.6 Character set and encoding

All source code files of the CAN driver are restricted to the ASCII character set. The files are encoded in UTF-8 format, with only the 7-bit subset (values 0x00 … 0x7F) being used.

# 2 Using the CAN driver

This chapter describes the steps to incorporate the CAN driver into your application.

## 2.1 Installation and prerequisites

*Note: Before you start, see the EB tresos Studio for ACG8 user's guide [10] for the following information.*

1. *The installation procedure of EB tresos ECU AUTOSAR components*
2. *The usage of the EB tresos Studio*
3. *The usage of the EB tresos ECU AUTOSAR build environment (It includes the steps to setup and integrate the own application within the EB tresos ECU AUTOSAR build environment)*

The installation of the CAN driver compiles with the general installation procedure for EB tresos ECU AUTOSAR components given in the documents mentioned above. If the driver has successfully been installed, it will appear in the module list of the EB tresos Studio (see *EB tresos Studio for ACG8 user's guide* [10]).

This document assumes that the project is property setup and using the application template as described in the *EB tresos Studio for ACG8 user's guide* [10]. This template provides the necessary folder structure, project and makefiles needed to configure and compile your application within the build environment. You must be familiar with the usage of the command shell.

All needed port pins need to be configured in the PORT driver to use digital input output functionality of the CAN driver. Check the *Specification of module PORT driver* [9] for PORT driver configuration.

## 2.2 Configuring the CAN driver

The CAN driver can be configured with any AUTOSAR-compliant generic configuration editor (GCE) tool. Save the configuration in a separate file named such as *Can.epc*. See chapter 4 EB tresos Studio configuration interface.

### 2.2.1 Architecture specifics

See section 4 EB tresos Studio configuration interface, for all vendor- and driver-specific configuration parameters.

## 2.3 Adapting your application

The relationship between the CAN bundle and other AUTOSAR modules is shown in Figure 3. The CAN driver is normally used via the CAN interface and may not be accessed directly. See the AUTOSAR *Specification of the CAN interface* [6], [8] for more information.

## 2.4 Starting the build process

Do the following to build your application:

*Note: For a clean build, use the build command with target `clean_all` before (`make clean_all`).*

1. Type the following command into the command shell to generate the necessary configuration dependent files. See 3.3 Generated files for details.

```
> make generate
```

2. Type the following command to resolve required file dependencies:

```
> make depend
```

3. Type the following command to compile and link the application:

```
> make (optional target: all)
```

The application is now built. All files are compiled and linked to a binary file which can be downloaded to the target hardware.

## 2.5 Measuring stack consumption

Do the following to measure stack consumption. It requires the Base module for proper measurement.

*Note: All files (including library files) should be rebuilt with the dedicated compiler option. The executable file built in this step must be used only to measure stack consumption.*

1. Add the following compiler option to the Makefile to enable stack consumption measurement.

```
-DSTACK_ANALYSIS_ENABLE
```

2. Type the following command to clean library files.

```
> make clean_lib
```

3. Follow the build process described in section 2.4 Starting the build process.
4. Follow the instructions in the release notes and measure the stack consumption.

## 2.6 Memory mapping

The *Can_MemMap.h* file in the *$(TRESOS_BASE)/plugins/MemMap_TS_T40D13M0I0R0/include* directory is a sample. This file is replaced by the file generated by MEMMAP module. Input to MEMMAP module is generated as *Can_Bswmd.arxml* in the *$(PROJECT_ROOT)/ output/generated/swcd* directory of your project folder.

### 2.6.1 Memory allocation keyword

- `CAN_START_SEC_CODE_ASIL_B / CAN_STOP_SEC_CODE_ASIL_B`

  The memory section is CODE. All executable code is allocated in this section

- `CAN_START_SEC_CONST_ASIL_B_UNSPECIFIED / CAN_STOP_SEC_CONST_ASIL_B_UNSPECIFIED`

  The memory section type is CONST. The following constants are allocated in this section:

  - CAN configuration data
  - Tx / Rx buffer element size
  - Data for converting between message length and DLC code

- `CAN_START_SEC_VAR_INIT_ASIL_B_UNSPECIFIED / CAN_STOP_SEC_VAR_INIT_ASIL_B_UNSPECIFIED`

  The memory section type is VAR. The following variable is allocated in this section:

  - Pointer to specified configuration setting
  - Store CAN controller initialization state
  - RxFIFO Ack notification information to CAN controller
  - Pointer to whole configuration setting

- `CAN_START_SEC_VAR_CLEARED_ASIL_B_UNSPECIFIED / CAN_STOP_SEC_VAR_CLEARED_ASIL_B_UNSPECIFIED`

  The memory section type is VAR. The following variable is allocated in this section:

  - Store Tx handle of PduId
  - Information for CAN driver state
  - Store Rx dedicated ID and Rx FIFO filters
  - Store wakeup message receive counter in pretended networking mode

  Store wakeup message receive counter in pretended networking mode

### 2.6.2 Assembler code

Assembler code for the CAN driver uses the fixed memory section names in Table 2.

**Table 2 Fixed section names**

| Section name | Allocate area |
|---|---|
| `Can_BitSearch_asm` | ROM area |

*Note: Depending on the compiler or its option, the assembler code may be defined as inline functions and not allocated in this section.*

# 3 Structure and dependencies

The CAN driver consists of static, configuration, and generated files.

## 3.1 Static files

- $(PLUGIN_PATH)=$(TRESOS_BASE)/plugins/Can_TS_* is the path to the CAN driver plugin.
- *$(PLUGIN_PATH)/lib_src* contains all static source files of the CAN driver. These files contain the functionality of the driver, which does not depend on the current configuration, with a part of exceptions. The files are grouped into a static library.
- *$(PLUGIN_PATH)/lib_include* contains all the internal header files for the CAN driver.
- *$(PLUGIN_PATH)/src* comprises configuration dependent source files or special derivative files. Each file will be built again when the configuration is changed.

All necessary source files will be automatically compiled and linked during the build process and all include paths will be set if the CAN driver is enabled.

- *$(PLUGIN_PATH)/include* is the basic public include directory that you need to include in *Can.h*.
- *$(PLUGIN_PATH)/autosar* directory contains the AUTOSAR ECU parameter definition with vendor, architecture, and derivative specific adaptations to create a correct matching parameter configuration for the CAN driver.

## 3.2 Configuration files

The configuration is done via EB tresos Studio software. The file containing the CAN driver's configuration is stored in *Can.xdm* and located in the *$(PROJECT_ROOT)/config* directory. This file serves as the input for the generation of the configuration-dependent source and header files during the build process.

## 3.3 Generated files

During the build process, the following files are generated based on the current configuration. They are in the *output/generated* sub folder of your project folder.

- include/Can_Cfg.h
- include/Can_ExternalInclude.h
- include/Can_PBcfg.h
- src/Can_Irq.c
- src/Can_PBcfg.c

Note: *You do not need to add the generated source files to your application make file; they are compiled and linked automatically during the build process.*

- swcd/Can_Bswmd.arxml

Note: *Additional steps are required for the generation of BSW module description. In EB tresos Studio, follow the menu path **Project** > **Build Project** and click **generate_swcd**.*

## 3.4 Dependencies

Figure 3 shows the relationship between the modules of the CAN bundle and other modules.

*Note: To use the CAN driver, you must enable and configure the PORT driver (see 3.4.1 PORT driver), the ECU state manager (EcuM) (see Specification of ECU state manager [3]), and the BSW scheduler module (see 3.4.7 BSW scheduler).*

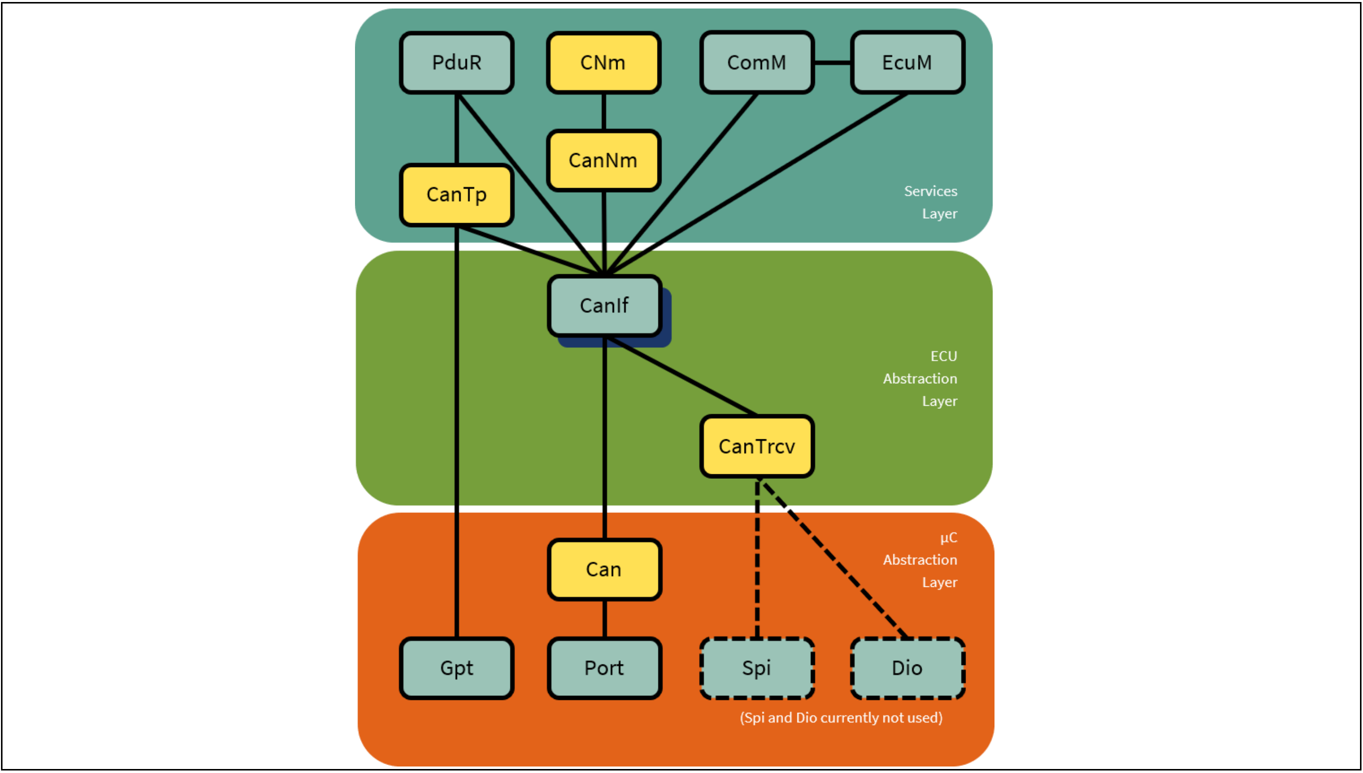

**Figure 3 Relationship between the CAN bundle and other AUTOSAR modules**

### 3.4.1 PORT driver

Although the CAN driver can be successfully compiled and linked without an AUTOSAR compliant PORT driver, the latter is required to configure and enable the port to be used. Otherwise, the CAN driver does not work as expected. The PORT driver needs to be initialized before the CAN driver is initialized. See the PORT driver's user guide for details.

### 3.4.2 MCU driver

The MCU driver needs to be initialized and all MCU clock reference points referenced by the CAN driver channels via configuration parameter `CanCpuClockRef` must have been activated (via calls of MCU API functions) before initializing the CAN driver. See the MCU driver's user guide for details.

### 3.4.3 CAN interface

The CAN interface is part of the ECU abstraction layer that is located above the CAN driver. It is the only module that calls the CAN driver functions and provides callback functions for the CAN driver events like transmit confirmation or receive indication.

### 3.4.4 DET

If default error detection is enabled in the CAN driver configuration, the DET needs to be installed, configured, and integrated into the application.

This driver reports DET error codes as instance 0.

### 3.4.5 DEM

If the DEM event report is enabled in the CAN driver configuration, the DEM needs to be installed, configured, and integrated into the application.

To enable DEM support in the CAN driver, the following production error needs to be defined in the DEM configuration in the container `CanDemEventParameterRefs`:

`CAN_E_HARDWARE_ERROR`

### 3.4.6 AUTOSAR OS

The AUTOSAR operating system handles the interrupts used by the CAN driver. See 6.3 Interrupts.

### 3.4.7 BSW scheduler

The BSW scheduler module handles the critical sections that are used by the CAN driver.

### 3.4.8 ECU state manager

The EcuM module handles the mode switches of the ECU.

### 3.4.9 Error callout handler

The error callout handler is called on every error that is detected, regardless of whether default error detection is enabled or disabled. The error callout handler is an ASIL safety extension that is not specified by AUTOSAR. It is configured via the `CanErrorCalloutFunction` configuration parameter.

### 3.4.10 Protocol error callout handler

The protocol error callout handler is used to notify the upper layer of protocol errors (Stuff error, Form error, Ack error, Bit1 error, Bit0 error, CRC error), status (ErrorActive, ErrorPassive, BusOff), and error counts (Tx/Rx Error count). The Protocol error callout handler is not specified by AUTOSAR. It is configured via the `CanProtocolErrorCalloutFunction` configuration parameter.

# 4 EB tresos Studio configuration interface

The GUI is not part of this delivery. For further information, see *EB tresos Studio for ACG8 user's guide* [10].

## 4.1 General configuration

The CAN driver configuration is described in the AUTOSAR ECU configuration parameter definition file. See this file for further information.

### 4.1.1 CanChangeBaudrateApi

**Description**

Controls the availability of the `Can_ChangeBaudrate()` API function.

**Annotation**

None

### 4.1.2 CanDevErrorDetection

**Description**

Switches the default error tracer (DET) detection and notification ON or OFF.

**Annotation**

Setting this parameter to FALSE disables the notification of development errors via DET. However, in contrast to the AUTOSAR specification, detection of development errors is still enabled and errors will be reported via `CanErrorCalloutFunction`.

### 4.1.3 CanHardwareCancellation

**Description**

Specifies whether to set the hardware transmission cancellation to ON or OFF.

**Annotation**

Cancellation is only possible for transmit messages using buffers.

Vendor-specific parameters.

### 4.1.4 CanIdenticalIdCancellation

**Description**

Enables or disables cancellation of pending PDUs with identical ID.

**Annotation**

Vendor-specific parameters.

### 4.1.5 CanIndex

**Description**

Specifies the instance Id of this module instance.

**Range**

0

**Annotation**

Use of multiple instances of the CAN driver is not supported.

### 4.1.6 CanLPduReceiveCalloutFunction

**Description**

This parameter defines the existence and the name of a callout function that is called after a successful reception of the message.

**Annotation**

This parameter defines the existence and the name of a callout function that is called after a successful reception of a received CAN Rx L-PDU (usually `CanIf_RxIndication()`). If this parameter is omitted (`#undef`), no callout takes place.

### 4.1.7 CanMainFunctionBusoffPeriod

**Description**

This parameter describes the period for cyclic call to `Can_MainFunction_Busoff()`.

**Range**

0.001 .. 65.535

**Annotation**

Cyclic period of the read main function in seconds.

### 4.1.8 CanMainFunctionModePeriod

**Description**

This parameter describes the period for cyclic call to `Can_MainFunction_Mode()`.

**Range**

0.001 .. 65.535

**Annotation**

Cyclic period of the read main function in seconds.

### 4.1.9 CanMainFunctionWakeupPeriod

**Description**

This parameter describes the period for cyclic call to `Can_MainFunction_Wakeup()`.

**Range**

0.001 .. 65.535

**Annotation**

Cyclic period of the read main function in seconds.

### 4.1.10 CanMultiplexedTransmission

**Description**

Specifies whether to set the multiplexed transmission to ON or OFF.

**Annotation**

When multiplexed transmission is enabled, the `CanHwObjectCount` parameter of transmission objects can be set to values greater than 1. Such objects allow multiple calls of `Can_Write()` before transmission of the first message is indicated. They implement a priority queue for messages.

### 4.1.11 CanPublicIcomSupport

**Description**

Specifies whether to set the pretended networking mode to ON or OFF.

**Annotation**

None

### 4.1.12 CanRxIndicationCompatibility

**Description**

Switches the interface of `CanIf_RxIndication()` according to ASR release.

**Range**

`CAN_ASR_422_COMPATIBILITY`: Call `CanIf_RxIndication()` of AUTOSAR 4.2.2.

`CAN_ASR_403_COMPATIBILITY`: Call `CanIf_RxIndication()` of AUTOSAR 4.0.3.

**Annotation**

Vendor-specific parameters

### 4.1.13 CanSetBaudrateApi

**Description**

Controls the availability of the `Can_SetBaudrate()` API function.

**Annotation**

None

### 4.1.14 CanSetRxFilterApi

**Description**

Controls the availability of the `Can_SetRxFilter()` API function.

**Annotation**

Vendor-specific parameters.

### 4.1.15 CanSetBaudrateAnyParametersApi

**Description**

Controls the availability of the `Can_SetBaudrateAnyParameters()` API function.

**Annotation**

Vendor-specific parameters.

### 4.1.16 CanSilentMode

**Description**

Controls the availability of the `Can_SetHwOperationMode()` API function.

**Annotation**

Vendor-specific parameters.

### 4.1.17 CanFlushApi

**Description**

Controls the availability of the `Can_Flush()` API function.

**Annotation**

Vendor-specific parameters.

### 4.1.18 CanTimeoutDuration

**Description**

Specifies the maximum time for blocking function until a timeout is detected. Unit is seconds.

**Range**

0.000001 .. 65.535

**Annotation**

`CanTimeoutDuration` must be greater than or equal to the value obtained from `OsSecondsPerTick` of the Os module specified in `CanOsCounterRef`.

### 4.1.19 CanVersionInfoApi

**Description**

Controls the availability of the `Can_GetVersionInfo()` API function.

**Annotation**

None

### 4.1.20 CanOsCounterRef

**Description**

Contains a reference to `OsSecondsPerTick` in the `OsCounter` container used by the CAN driver.

**Annotation**

None

### 4.1.21 CanSupportTTCANRef

**Description**

See `CanIfSupportTTCAN` parameter in the CAN interface module configuration.

**Annotation**

The driver does not support TTCAN, so the parameter can be omitted.

### 4.1.22 CanIcomLevel

**Description**

Defines the level of pretended networking.

**Annotation**

All ON/OFF for pretended networking mode are controlled only by `CanPublicIcomSupport`. So, the parameter can be omitted.

### 4.1.23 CanIcomVariant

**Description**

Defines the variant, which is supported by this CAN controller

**Annotation**

All ON/OFF for pretended networking mode are controlled only by `CanPublicIcomSupport`. So, the parameter can be omitted.

### 4.1.24 CanMainFunctionReadPeriod

**Description**

Describes the period for cyclic call to `Can_MainFunction_Read()`.

**Range**

0.001 .. 65.535

**Annotation**

Cyclic period of the read main function in seconds.

### 4.1.25 CanMainFunctionWritePeriod

**Description**

Describes the period for cyclic call to `Can_MainFunction_Write()`.

**Range**

0.001 .. 65.535

**Annotation**

Cyclic period of the write main function in seconds.

### 4.1.26 CanErrorCalloutFunction

**Description**

The error callout function is called on every error. The ASIL level of this function limits the ASIL level of the CAN driver.

Syntax:

```
  void ErrorCalloutHandler
(
uint16 ModuleId,
uint8 InstanceId,
uint8 ApiId,
uint8 ErrorId
) ;
```

**Annotation**

Vendor-specific parameters

### 4.1.27 CanProtocolErrorCalloutFunction

**Description**

The protocol error callout function is called when the error state changes . Available only if the `CanProtocolError` parameter is enabled. To use this callout function, a dedicated include file name must be added to `CanIncludeFile`.

Syntax:

```
void ProtocolErrorCalloutHandler
(
uint8 Controller,
uint8 Status,
uint8 TxErrorCnt,
uint8 RxErrorCnt,
uint8 ErrorReason
)
```

**Annotation**

Vendor-specific parameters

### 4.1.28 CanGetStatusApi

**Description**

Controls the availability of the `Can_GetStatus()` API function.

**Annotation**

Vendor-specific parameters

### 4.1.29 CanDeInitApi

**Description**

Controls the availability of the `Can_DeInit()` API function.

**Annotation**

Vendor-specific parameters

### 4.1.30 CanSetBaudrateInChangedClockApi

**Description**

Controls the availability of the `Can_SetBaudrateInChangedClock()` API function.

**Annotation**

Vendor-specific parameters

### 4.1.31 CanDemEventParameterRefs

**Description**

Container for the references to `DemEventParameter` elements, which will be invoked using the `Dem_ReportErrorStatus` API in case the corresponding error occurs. The EventId is taken from the referenced `DemEventParameter's DemEventId` value. The standardized errors are provided in the container and can be extended by vendor-specific error references.

**Annotation**

When using this parameter, define error in `DemEventParameter`.

Vendor-specific parameters.

## 4.2 Can settings configuration

The `CanConfigSet` is not set by default. This parameter can be set to a maximum of 10.

### 4.2.1 CanController

#### 4.2.1.1 General

##### 4.2.1.1.1 CanBusoffProcessing

**Description**

Enables or disables the `Can_MainFunction_BusOff()` API for handling bus-off and ECC error events in polling mode.

**Range**

- `INTERRUPT`: When bus-off or ECC error occur, the calling `Can_MainFunction_BusOff()` API cannot detect it.
- `POLLING`: When bus-off or ECC error occur, the calling `Can_MainFunction_BusOff()` API can detect it.

**Annotation**

None

##### 4.2.1.1.2 CanControllerActivation

**Description**

Defines if a CAN controller is used in the configuration.

**Annotation**

Activated controllers must have lower `CanControllerIds` when compared to deactivated controllers.

##### 4.2.1.1.3 CanControllerBaseAddress

**Description**

Only one address corresponding to the controller configured as `CanControllerPhysicalChannel` is allowed.

**Range**

0 .. 4294967295

**Annotation**

The address is automatically calculated by the channel selected by `CanControllerPhysicalChannel`.

#### 4.2.1.1.4 CanControllerId

**Description**

This parameter provides the controller ID, which is unique in each CAN driver.

**Range**

0 .. 255

**Annotation**

The value for this parameter starts with 0 and continues sequentially. If `CanControllerActivation` is set to FALSE, this ID is not known and may result in a gap. Controller IDs of deactivated controllers must be greater than the controller IDs of the activated controllers.

#### 4.2.1.1.5 CanRxProcessing

**Description**

Enables or disables the `Can_MainFunction_Read()` API for handling PDU reception events in polling mode.

**Range**

- `INTERRUPT`: When receiving a message, reception processing is not performed even if `Can_MainFunction_Read()` is called.
- `POLLING`: When receiving a message, it performs reception processing by calling `Can_MainFunction_Read()`.

**Annotation**

None

#### 4.2.1.1.6 CanTxProcessing

**Description**

Enables or disables the `Can_MainFunction_Write()` API from handling PDU transmission events in polling mode.

**Range**

- `INTERRUPT`: When transmission is completed, even if `Can_MainFunction_Write()` is called, transmission completion processing is not performed.
- `POLLING`: When transmission is completed, `Can_MainFunction_Write()` is called to perform transmission completion processing.

**Annotation**

None

#### 4.2.1.1.7 CanWakeupFunctionalityAPI

**Description**

Controls the availability of the `Can_CheckWakeup()` API function.

**Annotation**

AUTOSAR specifies this parameter to be part of the CAN controller container, but it has global scope. This means that the `Can_CheckWakeup()` function is switched OFF only if this parameter is disabled in all CAN controller containers.

#### 4.2.1.1.8 CanWakeupProcessing

**Description**

Enables or disables the `Can_MainFunction_Wakeup()` API for handling wakeup events in polling mode.

**Range**

- `INTERRUPT`: When a wake up event occurs while CAN controller is in Sleep state, Wake up does not occur even if `Can_MainFunction_Wakeup()` is called.
- `POLLING`: When a wake up event occurs while the CAN controller is in Sleep state, it calls `Can_MainFunction_Wakeup()` to wake up.

**Annotation**

None

#### 4.2.1.1.9 CanWakeupSupport

**Description**

Keeps the CAN controller hardware in bus monitoring mode and detects wakeup upon reception of any frame configured by the filter mask settings.

The CAN controller's power domain, clock, and the transceiver must be kept active for that purpose. Wakeup can be stopped by disabling one of these items. It can also be stopped by keeping the controller in Stop mode instead of Sleep mode.

**Annotation**

Enable only

#### 4.2.1.1.10 CanControllerInstance

**Description**

Selects the CAN controller instance on the target.

**Range**

The number of supported CAN controllers depends on the microcontroller.

**Annotation**

Select this parameter and set the channel to `CanControllerPhysicalChannel`.

Vendor-specific parameters

#### 4.2.1.1.11 CanControllerPhysicalChannel

**Description**

Selects the physical CAN controller on the target.

**Range**

The number of supported CAN controllers depends on the microcontroller.

**Annotation**

Vendor-specific parameters

#### 4.2.1.1.12 CanMessageRamBaseAddress

**Description**

Refers to the base address (for CPU access) of the message RAM that is used by the CAN controller. The physical base address must be used, regardless of whether the full RAM is used by the controller.

**Range**

The number of supported CAN controllers depends on the microcontroller.

**Annotation**

The address is automatically calculated by the instance selected by `CanControllerInstance`.

Also, the message RAM is shared by the same instance. So even if values overlap, there is no problem.

Vendor-specific parameters

#### 4.2.1.1.13 CanMessageRamSize

**Description**

Indicates the size in bytes of the message RAM that is reserved for this CAN controller beginning from `CanMessageRamBaseAddress`.

**Range**

The number of supported CAN controllers depends on the microcontroller.

**Annotation**

The address is automatically calculated by the instance selected by `CanControllerInstance`.

Also, the message RAM is shared by the same instance. So even if values overlap, there is no problem.

Vendor-specific parameters

#### 4.2.1.1.14 CanProtocolError

**Description**

Controls the availability of the protocol error detection function. The name of the function to be called can be configured by the `CanProtocolErrorCalloutFunction` parameter.

**Annotation**

For this parameter to be effective, you must define a callout function in CanProtocolErrorCalloutFunction.

Vendor-specific parameters.

#### 4.2.1.1.15 CanControllerDefaultBaudrate

**Description**

Contains a reference to baudrate configuration container configured for the CAN controller.

**Annotation**

None

#### 4.2.1.1.16 CanCpuClockRef

**Description**

Contains a reference to the CPU clock configuration, which is set in the MCU driver configuration.

**Annotation**

Set the clock with the MCU driver, in advance, according to the channel to be used.

#### 4.2.1.1.17 CanWakeupSourceRef

**Description**

Contains a reference to the Wakeup source for this controller as defined in the ECU state manager.

**Annotation**

`CanWakeupSourceRef` refers to the `EcuMWakeupSource` of the EcuM module. If there is no reference destination, the Wakeup source Id is set to 1.

#### 4.2.1.1.18 CanTTController

**Description**

Sets parameters of the TTCAN controller.

**Annotation**

The CAN driver does not support TTCAN. Therefore, the container `CanTTController` and all its internals are not available.

#### 4.2.1.2 CanControllerBaudrateConfig

##### 4.2.1.2.1 CanControllerBaudRate

**Description**

Specifies the baudrate of the controller in kbps.

**Range**

1 .. 2000

**Annotation**

Measured in kbps. 0 is not allowed, other values may also not be allowed depending on the clock settings.

When `CanChangeBaudrateApi` is enabled, the value of this parameter should not be duplicated.

##### 4.2.1.2.2 CanControllerBaudRateConfigID

**Description**

Uniquely identifies a specific baud rate configuration. This ID is used by `Can_SetBaudrate()` API.

**Range**

0 .. 65535

**Annotation**

None

##### 4.2.1.2.3 CanControllerPropSeg

**Description**

Specifies propagation delay in time quanta.

**Range**

0

**Annotation**

The `M_TTCAN` hardware does not support a propagation time. Instead, the time segment 1 must be increased accordingly.

##### 4.2.1.2.4 CanControllerSeg1

**Description**

Specifies phase segment 1 in time quanta.

**Range**

2 .. 255

**Annotation**

Range is limited due to the `M_TTCAN` hardware constraints.

#### 4.2.1.2.5 CanControllerSeg2

**Description**

Specifies phase segment 2 in time quanta.

**Range**

2 .. 128

**Annotation**

Range is limited due to the `M_TTCAN` hardware constraints.

#### 4.2.1.2.6 CanControllerSyncJumpWidth

**Description**

Specifies the synchronization jump width for the controller in time quanta.

**Range**

1 .. 128

**Annotation**

Range is limited due to the `M_TTCAN` hardware constraints.

### 4.2.1.3 CanControllerFdBaudrateConfig

#### 4.2.1.3.1 CanControllerFdBaudRate

**Description**

Specifies the data segment baud rate of the controller in kbps.

**Range**

1 .. 16000

**Annotation**

Measured in kbps. 0 is not allowed, other values may also not be allowed depending on the clock settings.

#### 4.2.1.3.2 CanControllerPropSeg

**Description**

Specifies propagation delay in time quanta of the data section.

**Range**

0

**Annotation**

The M_TTCAN hardware does not support a propagation time. Instead, the time segment 1 must be increased accordingly.

#### 4.2.1.3.3 CanControllerSeg1

**Description**

Specifies phase segment 1 in time quanta of the data section.

**Range**

1 .. 32

**Annotation**

Range is limited due to the `M_TTCAN` hardware constraints.

#### 4.2.1.3.4 CanControllerSeg2

**Description**

Specifies phase segment 2 in time quanta of the data section.

**Range**

1 .. 16

**Annotation**

Range is limited due to the `M_TTCAN` hardware constraints.

#### 4.2.1.3.5 CanControllerSyncJumpWidth

**Description**

Specifies the synchronization jump width for the controller in time quanta of the data section.

**Range**

1 .. 16

**Annotation**

Range is limited due to the `M_TTCAN` hardware constraints.

#### 4.2.1.3.6 CanControllerTrcvDelayCompensationOffset

**Description**

Specifies the transceiver delay compensation offset in ns. If not specified, transceiver delay compensation is disabled.

Example:

- Condition
  - The secondary sample point (SSP) = 80%
  - CAN FD baud rate (CanControllerFdBaudRate) = 2.5 Mbps
    FD BitTime = 1/(2.5 * 10^6) s/Bit = 0.4 * 10^(-6) = 400 ns

- Calculation formula

  CanControllerTrcvDelayCompensationOffset = (SSP%) * FD BitTime = 80% * 400 ns = 320 ns

**Range**

0 .. 400

**Annotation**

The delay time is rounded to a divided by a frequency referred by `CanCpuClockRef`.

#### 4.2.1.3.7 CanControllerTxBitRateSwitch

**Description**

Specifies whether the bit rate switching will be used for transmissions. If FALSE: CAN-FD frames will be sent without bit rate switching.

**Annotation**

None

### 4.2.2 CanHardwareObject

#### 4.2.2.1 General

##### 4.2.2.1.1 CanFdPaddingValue

**Description**

Specifies the value which is used to pad unspecified data in CAN-FD frames is greater than 8 bytes for transmission. This is necessary due to the discrete possible values of the DLC is greater than 8 bytes. If the length of a PDU, which was requested to be sent does not match the allowed DLC values, the remaining bytes up to the next possible value will be padded with this value.

**Range**

0 .. 255

**Annotation**

When `CanObjectType` is `TRANSMIT`, you can enter a value.

##### 4.2.2.1.2 CanHandleType

**Description**

Specifies the type of a hardware object.

**Range**

*BASIC*: For several L-PDUs are handled by the hardware object

*FULL*: For only one L-PDU (identifier) is handled by the hardware object

**Annotation**

it has influence on CanIf only, but not on CAN.

##### 4.2.2.1.3 CanHwObjectCount

**Description**

The parameter controls the size of message queue in number of messages.

**Range**

1 .. 64

**Annotation**

Dedicated message objects must have a value of 1.

#### 4.2.2.1.4 CanIdType

**Description**

Specifies the type of IdValue

**Range**

`STANDARD`: All the CANIDs are of type standard only (11 bit).

`EXTENDED`: All the CANIDs are of type extended only (29 bit).

`MIXED`: The type of CANIDs can be both Standard or Extended.

**Annotation**

`EXTENDED`, `STANDARD` for receiving messages,

`EXTENDED`, `MIXED`, `STANDARD` for transmitting messages.

Mixed mode is not supported for reception.

#### 4.2.2.1.5 CanObjectId

**Description**

Holds the handle ID of HRH or HTH.

**Range**

0 .. 65535

**Annotation**

The value of this parameter is unique in each CAN driver, and it should start with 0 and continue sequentially. Furthermore, the `CanObjectIds` of HRHs must be smaller than those of HTHs.

#### 4.2.2.1.6 CanIdValue

**Description**

Specifies (together with the filter mask) the identifiers range that passes the hardware filter.

**Range**

0x0 .. 0x7FF for standard CAN IDs

0x0 .. 0x1FFFFFFF for extended CAN IDs

**Annotation**

None

#### 4.2.2.1.7 CanObjectType

**Description**

Specifies if the `HardwareObject` is used as transmit or as receive object.

**Range**

`RECEIVE`: Receives HOH

`TRANSMIT`: Transmits HOH

**Annotation**

The selection between dedicated buffer and queue for transmit messages is done via the `CanHwObjectCount` configuration parameter.

#### 4.2.2.1.8 CanRxBufferSelection

**Description**

Selects the allocation of `CanHwObject` to a physical RX buffer type. Selection of `CanHwObjects` to TX buffers is done via configuration parameter `CanHwObjectCount`.

**Range**

`CAN_RX_DEDICATED`: Stores the received CAN message in RX-Buffer.

`CAN_RX_FIFO0`: Stores the received CAN message in RXFIFO-0.

`CAN_RX_FIFO1`: Stores the received CAN message in RXFIFO-1.

**Annotation**

Vendor-specific parameters

#### 4.2.2.1.9 CanTriggerTransmitEnable

**Description**

Defines whether CAN supports the trigger-transmit API for this handle.

**Annotation**

If the `CanTriggerTransmitEnable` has enabled, `CanIf_TriggerTransmit` operation is possible to call.

#### 4.2.2.1.10 CanControllerRef

**Description**

Reference to CAN controller to which the HOH is associated to.

**Annotation**

None

#### 4.2.2.1.11 CanMainFunctionRWPeriodRef

**Description**

Reference to `CanMainFunctionReadPeriod` and `CanMainFunctionWritePeriod`

**Annotation**

None

#### 4.2.2.1.12 CanIcomRxMessageDedicated

**Description**

Defines whether this handle is dedicated to `CanIcomRxMessage`.

**Annotation**

When this parameter is enabled, CanIcom's `CanIcomMessageId` and `CanIcomMessageIdMask` filter settings are used. Only RxFIFO0 can be set for filter setting of ICOM.

Vendor-specific parameters.

### 4.2.2.2 CanHwFilter

#### 4.2.2.2.1 CanHwFilterCode

**Description**

Specifies (together with the filter mask) the identifiers range that passes the hardware filter.

**Range**

0x0 .. 0x7FF for standard CAN IDs

0x0 .. 0x1FFFFFFF for extended CAN IDs

**Annotation**

None

#### 4.2.2.2.2 CanHwFilterMask

**Description**

The CAN identifiers of incoming messages are appropriately masked.

**Range**

0x0 .. 0x7FF for standard CAN IDs

0x0 .. 0x1FFFFFFF for extended CAN IDs

**Annotation**

If a bit is set to '0', it means that the corresponding bit position of the CAN ID is not checked during hardware filtering.

#### 4.2.2.3 CanTTHardwareObjectTrigger

**Description**

`CanTTHardwareObjectTrigger` is specified in the SWS TTCAN and contains the configuration (parameters) of TTCAN triggers for hardware objects, which are additional to the configuration (parameters) of CAN hardware objects.

**Annotation**

The CAN driver does not support TTCAN. Therefore, the `CanTTHardwareObjectTrigger` container and all its internals are not available.

### 4.2.3 CanIcom

#### 4.2.3.1 CanIcomConfig

##### 4.2.3.1.1 CanIcomConfigId

**Description**

Identifies the ID of the ICOM configuration.

**Range**

1 .. 255

**Annotation**

The value of this parameter is unique in each ICOM configuration, and it should start with 1 and continue sequentially.

##### 4.2.3.1.2 CanIcomWakeOnBusOff

**Description**

Defines whether the MCU will wake if the bus-off is detected in the pretended networking mode.

**Annotation**

The software ICOM supported CAN driver is prohibited to transmit in pretended networking mode. Therefore, this parameter is not supported.

### 4.2.3.2 CanIcomRxMessage

#### 4.2.3.2.1 CanIcomCounterValue

**Description**

Defines the counter value that will wake up the MCU when the message with the ID is received, the specified number of times, on the communication channel in the pretended networking mode.

**Range**

1 .. 65535

**Annotation**

The value of this parameter is unique in each ICOM configuration, and it should start with 1 and continue sequentially.

#### 4.2.3.2.2 CanIcomMessageId

**Description**

Defines the message ID that causes `CanIcomRxMessage` to wake up. In addition, a mask (`CanIcomMessageIdMask`) can be defined. In that case, it is possible to define a range of RX messages, which can create a wakeup condition.

**Range**

0x0 ... 0x7FF for standard CAN IDs

0x0 ... 0x1FFFFFFF for extended CAN IDs

**Annotation**

None

#### 4.2.3.2.3 CanIcomMessageIdMask

**Description**

Masks the CAN identifiers of incoming messages. If the masked identifier matches the masked value of `CanIcomMessageId`, it can create a wakeup condition for `CanIcomRxMessage`. Bits holding a 0 mean that the message's identifier in the respective bit position do not need to be compared. The mask will be built by filling with leading 0.

**Range**

0x0 ... 0x7FF for standard CAN IDs

0x0 ... 0x1FFFFFFF for extended CAN IDs

**Annotation**

`CanIcomMessageIdMask` and `CanIcomRxMessageSignalConfig` cannot be mixed.

#### 4.2.3.2.4 CanIcomMissingMessageTimerValue

**Description**

Defines whether the MCU will wake if the message with the ID is not received for a specific time, in ms, on the communication channel in the pretended networking mode.

**Range**

0 .. 4294967295

**Annotation**

None

#### 4.2.3.2.5 CanIcomPayloadLengthError

**Description**

Defines whether the MCU will wake if a payload error occurs.

**Annotation**

None

#### 4.2.3.2.6 CanHardwareObjectRef

**Description**

References HOH to which the `CanIcomRxMessage` is associated.

**Annotation**

Can associate only HOH whose `CanIcomRxMessageDedicated` in `CanHardwareObject` is enabled. It is not possible to associate the same `CanHardwareObject` from each `CanIcomRxMessage`.

Vendor-specific parameters.

### 4.2.3.3 CanIcomRxMessageSignalConfig

#### 4.2.3.3.1 CanIcomSignalMaskH

**Description**

Masks a signal in the payload of a CAN message (Upper 32 bits).

**Range**

0 .. 4294967295

**Annotation**

Generates 64-bit data with the following combination:

Upper 32 bits (`CanIcomSignalMaskH`) + Lower 32 bits (`CanIcomSignalMaskL`)

Vendor-specific parameters.

#### 4.2.3.3.2 CanIcomSignalMaskL

**Description**

Masks a signal in the payload of a CAN message (Lower 32 bits).

**Range**

0 .. 4294967295

**Annotation**

Generates 64-bit data with the following combination:

Upper 32 bits (`CanIcomSignalMaskH`) + Lower 32 bits (`CanIcomSignalMaskL`)

Vendor-specific parameters.

#### 4.2.3.3.3 CanIcomSignalOperation

**Description**

Defines the operation, which will be used to verify whether the signal value creates a wakeup condition.

**Range**

- `AND`: If the masked payload via `CanIcomSignalMask` `AND` `CanIcomSignalValue` is TRUE, the MCU wakes up.
- `EQUAL`: If the masked payload via `CanIcomSignalMask` `EQUAL` `CanIcomSignalValue` is TRUE, the MCU wakes up.
- `GREATER`: If the masked payload via `CanIcomSignalMask` is lesser than `CanIcomSignalValue` is TRUE, the MCU wakes up.
- `SMALLER`: If the masked payload via `CanIcomSignalMask` is greater than `CanIcomSignalValue` is TRUE, the MCU wakes up.
- `XOR`: The masked payload via `CanIcomSignalMask` `XOR` `CanIcomSignalValue` is TRUE, the MCU wakes up.

**Annotation**

None

#### 4.2.3.3.4 CanIcomSignalValueH

**Description**

Defines a signal value which will be compared with the masked `CanIcomSignalMask` value of the received signal (Upper 32 bits). See `CanIcomSignalOperation` for comparison operation.

**Range**

0 .. 4294967295

**Annotation**

Generates 64-bit data with the following combination:

Upper 32 bits (`CanIcomSignalValueH`) + Lower 32 bits (`CanIcomSignalValueL`)

Vendor-specific parameters.

#### 4.2.3.3.5 CanIcomSignalValueL

**Description**

Defines a signal value which will be compared with the masked `CanIcomSignalMask` value of the received signal (Lower 32 bits). See `CanIcomSignalOperation` for comparison operation.

**Range**

0 .. 4294967295

**Annotation**

Generates 64-bit data with the following combination:

Upper 32 bits (`CanIcomSignalValueH`) + Lower 32 bits (`CanIcomSignalValueL`)

Vendor-specific parameters.

#### 4.2.3.3.6 CanIcomSignalRef

**Description**

Defines a reference to the signal which will be checked in addition to the message id (`CanIcomMessageId`).

**Annotation**

This parameter refers to the COM module's `ComSignal/ComFilter`. However, since it is possible to substitute the mask setting with the parameters of `CanIcomMessageSignal`, this parameter does not support.

### 4.2.4 CanIncludeFile

#### 4.2.4.1 CanIncludeFile

**Description**

Lists the file names that will be included in *Can_ExternalInclude.h*. Any application specific symbol that is used by the CAN configuration (e.g., Error callout function) should be included by configuring this parameter.

**Annotation**

Vendor-specific parameters

## 4.3 Implementation constants

The CAN driver configuration files are generated based on the parameter configuration that you provide. These include CAN driver internal data and data types.

The configuration variant is “post build time”. Therefore, the configuration is variant with respect to the parameter provided to `Can_Init`.

The identifier `<CanConfigSet container name>` can be used as parameter to `Can_Init`.

## 4.4 Other modules

### 4.4.1 PORT driver

The pins given in section 6.1 Ports and pins must be configured in the PORT driver.

### 4.4.2 MCU driver

The clock frequency supplied to the CAN driver must be set and initialized using the MCU driver.

### 4.4.3 CAN interface

The CAN interface must be configured to match the CAN driver's configuration.

### 4.4.4 DET

DET must be configured if default error detection is activated.

### 4.4.5 DEM

DEM must be configured if diagnostic event manager is activated.

### 4.4.6 AUTOSAR OS

The CAN driver's interrupts (listed in 6.3 Interrupts) must be configured in the AUTOSAR operating system.

*Note:* *The AUTOSAR OS must only configure those interrupts which are used by the CAN driver.*

### 4.4.7 BSW scheduler

The CAN driver uses the following services of the BSW scheduler (SchM) to enter and leave critical sections:

- `SchM_Enter_Can_CAN_EXCLUSIVE_AREA_0(void)`
- `SchM_Exit_Can_CAN_EXCLUSIVE_AREA_0(void)`

You must ensure that the BSW scheduler is properly configured and initialized before using the CAN services.

# 5 Functional description

The CAN driver is intended to provide a hardware-independent interface for the CAN interface to transmit and receive CAN messages and provide notifications for the special events, “bus-off” and “wakeup” over a CAN bus.

*Note:* *The CAN driver is usually used via the CAN interface (compare section 2.3 Adapting your application) and therefore its functions should not directly be called by the application. CAN driver functions are called exclusively by the CAN interface.*

## 5.1 Function of the module

### 5.1.1 CAN driver state machine

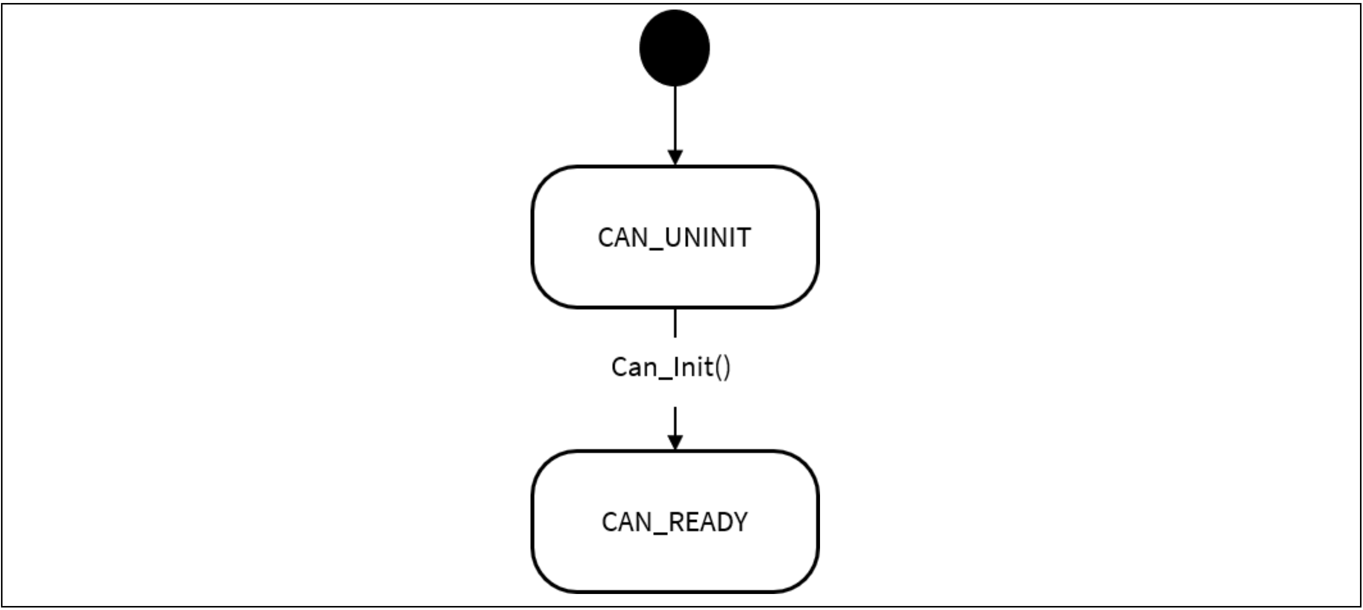

**Figure 4 CAN driver state machine**

The CAN driver's state machine is shown in Figure 4.

#### 5.1.1.1 State CAN_UNINIT

After power ON, the CAN driver is in the `CAN_UNINIT` state in which the driver has not been initialized yet.

#### 5.1.1.2 State CAN_READY

When the `CAN_READY` state is reached, the CAN driver is initialized and is ready to be used.

#### 5.1.1.3 State transitions

`CAN_UNINIT` is exited by calling `Can_Init()` which initializes the driver and performs a transition to the `CAN_READY` state.

*Note:* *The `CAN_UNINIT` state can only be exited via the `Can_Init()` function. This transition must take place before the CAN driver is used.*

### 5.1.2 CAN controller state machine

For each CAN controller, the CAN interface has a state machine. If default error detection is activated, the CAN driver simulates this state machine for error checks. The CAN driver's view of this state machine is given in Figure 5.

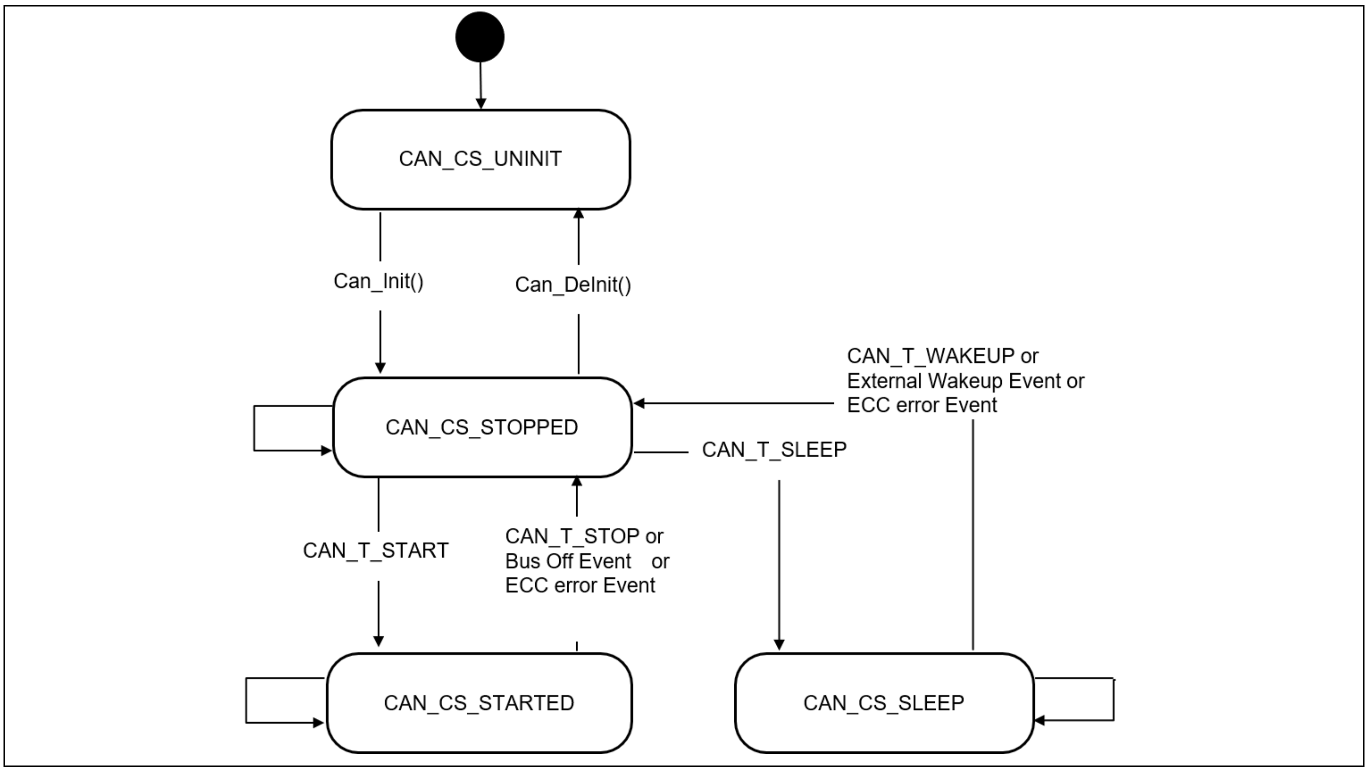
CAN_CS_UNINIT
Can_Init()
Can_DeInit()
CAN_T_WAKEUP or External Wakeup Event or ECC error Event
CAN_CS_STOPPED
CAN_T_SLEEP
CAN_T_START
CAN_T_STOP or Bus Off Event or ECC error Event
CAN_CS_STARTED
CAN_CS_SLEEP

**Figure 5 CAN controller state machine**

#### 5.1.2.1 State CAN_CS_UNINIT

`CAN_CS_UNINIT` is the state of the controllers after power on. The controllers have not been initialized yet.

#### 5.1.2.2 State CAN_CS_STOPPED

In the `CAN_CS_STOPPED` state, the CAN controller was initialized, but does not take part in the bus communication.

#### 5.1.2.3 State CAN_CS_STARTED

A controller in the `CAN_CS_STARTED` state is initialized and takes part in the communication on the bus.

#### 5.1.2.4 State CAN_CS_SLEEP

The controller listens to messages on the bus, but does not send anything (that is, it does not send ACK or error frames). The same message filters are active as in the `CAN_CS_STARTED` state. The controller will signal wakeup when any message is received. The content of the wakeup message is not passed to `CanIf`; instead it is withdrawn.

The following conditions are prerequisites for wakeup:

- The CAN controller's power domain is powered.
- The CAN controller's clock is active.
- The CAN transceiver is operational.

Note: *If the RAM that the CAN driver is linked to is cleared, then the CAN driver is not operational anymore. The CAN driver must be re-initialized in that case. Upon initialization, the driver recognizes the wakeup event that was caused before initialization.*

#### 5.1.2.5 State transitions

State transitions can be triggered in four ways:

1. If the CAN driver has not been initialized, a call to `Can_Init()` performs a transition to `CAN_CS_STOPPED`.
2. Among the two states `CAN_CS_STOPPED` and `CAN_CS_SLEEP`, transition to `CAN_CS_UNINIT` can be performed by calling `Can_DeInit()`.
3. Among the three states `CAN_CS_STOPPED`, `CAN_CS_STARTED`, and `CAN_CS_SLEEP`, transitions can be performed by calling `Can_SetControllerMode()` with the corresponding parameter (for example, `CAN_T_START` for a transition from `CAN_CS_STOPPED` to `CAN_CS_STARTED`).
4. Asynchronous events can trigger a transition. These events are a bus-off event and ECC error event in the `CAN_CS_STARTED` state, which triggers a transition to `CAN_CS_STOPPED`. In addition, external wakeup and ECC error events in `CAN_CS_SLEEP` state also trigger a transition to `CAN_CS_STOPPED`.

Figure 5 shows the different supported transitions.

### 5.1.3 Transmitting and receiving CAN messages

The main function of the CAN driver is to send and receive CAN messages and to notify the upper layer (CAN interface). This can only be performed in the `CAN_CS_STARTED` state. See section 5.1.1 CAN driver state machine, section 5.1.2 CAN controller state machine, and especially section 5.2 Initialization on how to reach this state.

A message is transmitted using the `Can_Write()` API function. This function checks the given parameters (compare section 7.4.6 Can_Write) and triggers a transmission on the hardware if the parameters are accepted.

For receiving messages, the CAN driver does not provide an API function. Instead the upper layer must provide a callback function as configured with `CanLPduReceiveCalloutFunction` (usually `CanIf_RxIndication()`). Whenever a message is received, it is provided to the upper layer via this interface (see section 5.1.4 Notifications for additional information).

### 5.1.4 Notifications

The CAN driver supports the following six different types of notification functions for the upper layer, which are called if one of the supported events occurs:

- **TX confirmation** after a successful message transmission
- **TX cancel confirmation** after a message was canceled instead of transmission
- **RX indication** after a successful message reception
- **Bus-off notification** after a bus-off event happened
- **Wakeup notification** when the controller was woken up from the CAN_CS_SLEEP state.
- **Protocol error notification** after a protocol error event happened

For all notifications and CAN controllers, you can select polling or interrupt mode. Depending on this configuration, the notification callback function provided by the upper layer (compare the section called `CanIf_TxConfirmation()`) is either called from the corresponding interrupt (in interrupt context!) or from the corresponding main function that is provided by the CAN driver, if polling is used.

These main functions (`Can_MainFunction_Write()`, `Can_MainFunction_Read()`, `Can_MainFunction_BusOff()`, `Can_MainFunction_Mode()`, and `Can_MainFunction_Wakeup()`) must be cyclically called to poll the corresponding event in order to invoke the upper layers callback function if polling is used.

*Note: Make sure that the call cycle is short enough so that events are not lost, but that it is long enough to keep the workload at an acceptable level.*

*Note: The functions can be called, even if interrupt mode is configured. In this case, the main function exits without action.*

For polling the RX indication, for example, make sure that at least one call of the `Can_MainFunction_Read()` is executed between two incoming messages. Otherwise, messages are lost and if the DET module is configured and enabled, a DET overflow error occurs and messages are lost.

*Note: An appropriate ISR in the AUTOSAR OS must be configured for all used interrupts. (See section 4.4 Other modules)*

### 5.1.5 Reception (RX) filter parameters

AUTOSAR 4.0.3 and AUTOSAR 4.2.2 specify different items to configure filter settings (See AUTOSAR specifications for details). The CAN driver implements both flavors of filter settings, which includes some redundant settings.

The filter settings are generated from the settings in the `CanHwFilter` container. At least one item of `CanHwFilter` is mandatory for each receive (RX) `CanHardwareObject`. Multiple filters can be applied to a single RX `CanHardwareObject`, which is assigned to an RX FIFO.

`CanIdValue` can be optionally used. If configured, this setting must match `CanHwFilterCode` in `CanHwFilter`.

## 5.2 Initialization

Initialization is done via the `Can_Init()` function, which results in the `CAN_CS_STOPPED` state.

To start the controller, call the `Can_SetControllerMode()` API function with the required transition (`CAN_T_START`).

An example initialization sequence that starts one controller can be performed by the following API calls:

```
Can_Init( CanConf_CanConfigSet_<name of CanConfigSet> );
Can_SetControllerMode( CanConf_CanController_<Id of CanConfigSet>_<name of
CanController>, CAN_T_START );
```

After initialization and start, messages can be sent by calling the `Can_Write()` function.

## 5.3 Runtime reconfiguration

Reconfiguring the whole CAN driver is possible with a `Can_DeInit()` function.

## 5.4 Runtime baud rate change

You can reconfigure each CAN controller separately, with a new baud rate by calling the `Can_ChangeBaudrate()` or `Can_SetBaudrate()` function while in the `CAN_CS_STOPPED` state.

*Note: The `Can_CheckBaudrate()` function can be used to check if a baud rate is supported.*

Also, by calling the `Can_SetBaudrateInChangedClock()` function in the `CAN_CS_STOPPED` state, it is possible to set the baud rate registered in configuration under the changed clock situation (each CAN controller can be individually reconfigured). You can also specify detailed baud rate parameters as arguments to the *`Can_SetBaudrateAnyParameters()`* function.

## 5.5 API parameter checking

The CAN driver's services perform regular error checks. When an error occurs, the error hook routine (configured via `CanErrorCalloutFunction`) is called and the error code, service ID, module ID, and instance ID are passed as parameters.

If default error detection is enabled, all errors are also reported to DET, the central error hook function within the AUTOSAR environment. The checking itself cannot be deactivated for safety reasons.

Table 3 shows the development error checks that are performed by the services of the CAN driver.

**Table 3 Development error codes**

| Related error code | Value | Type of error |
|---|---|---|
| `CAN_E_PARAM_POINTER` | 1 | API service was called with an invalid parameter. |
| `CAN_E_PARAM_HANDLE` | 2 | |
| `CAN_E_PARAM_DLC` | 3 | |
| `CAN_E_PARAM_CONTROLLER` | 4 | |
| `CAN_E_PARAM_BAUDRATE` | 8 | |
| `CAN_E_PARAM_ID` | 11 | |
| `CAN_E_PARAM_PDUID` | 15 | |
| `CAN_E_BAUDRATE_ANY_PARAM` | 18 | |
| `CAN_E_PARAM_RXFILTER` | 19 | |
| `CAN_E_PARAM_MODE` | 20 | |
| `CAN_E_UNINIT` | 5 | API service was used before the initialization of the module. |
| `CAN_E_TRANSITION` | 6 | Invalid transition for the current mode. |
| `CAN_E_DATALOST` | 7 | Received CAN message is lost. |
| `CAN_E_ICOM_CONFIG_INVALID` | 9 | API service called with wrong parameter: ICOM configuration ID. |
| `CAN_E_INIT_FAILED` | 10 | Invalid configuration set selection in `Can_Init()` function. |
| `CAN_E_UNKNOWN_DATA` | 12 | Received CAN message cannot be identified. |

| Related error code | Value | Type of error |
|---|---|---|
| `CAN_E_OS_TIME_REFUSED` | 13 | OS timer has refused its service. Timeout cannot be evaluated. |
| `CAN_E_IRQ_DEPTH` | 16 | Interrupt disable or enable violates nesting restrictions. |
| `CAN_E_CALC_PRESCALER` | 17 | The prescaler value calculated by the changed clock cannot be set. |

## 5.6 Production error detection

If ECC error occurs, `CAN_DEM_E_HW_ERROR` is reported to the DEM.

When an error occurs, the error hook routine (configured via `CanErrorCalloutFunction`) is also called and the error code (`CAN_E_UNCORRECTABLE_BIT_ERROR`), service ID, module ID, and instance ID are passed as parameters.

**Table 4 Production error codes**

| Related error code | Value | Type of error |
|---|---|---|
| `CAN_E_UNCORRECTABLE_BIT_ERROR` | 14 | Uncorrectable bit error in message RAM is detected. |

## 5.7 Reentrancy

See the reentrancy description of each API function in 7.4 Functions.

## 5.8 Debugging support

The CAN driver does not support debugging.

## 5.9 Execution time dependencies

The execution of the API function is dependent on certain factors. Table 5 lists these dependencies.

**Table 5 Execution time dependencies**

| Affected function | Dependency |
|---|---|
| `Can_Init()`<br>`Can_DeInit()`<br>`Can_MainFunction_Mode()`<br>`Can_MainFunction_Wakeup()`<br>`Can_SetHwOperationMode()` | Runtime depends on the number of configured controllers. |
| `Can_SetIcomConfiguration()` | Runtime depends on the number of configured controllers and configured ICOM. |
| `Can_SetControllerMode()` | Runtime depends on the number of RX and TX message objects configured for the corresponding controller and on the configuration of `CanTimeoutDuration`. |
| `Can_MainFunction_BusOff()` | Runtime depends on the number of controllers that are configured for polling mode. |
| `Can_ChangeBaudrate()`<br>`Can_CheckBaudrate()`<br>`Can_SetBaudrate()`<br>`Can_SetBaudrateInChangedClock()` | Runtime depends on the number of baud rates configured for the corresponding controller. |
| `Can_SetBaudrateAnyParameters()` | Runtime depends on the parameters specified. |
| `Can_MainFunction_Read()` | Runtime depends on the number of controllers that are configured for polling mode, the number of messages that have been received since the last call, and the runtime of `CanIf_RxIndication()`. |
| `Can_MainFunction_Write()` | Runtime depends on the number of controllers that are configured for polling mode, the number of messages that have |

| Affected function | Dependency |
|---|---|
| | been sent since the last call, the number of messages that have been canceled since the last call, and the runtime of `CanIf` callback functions that correspond to each event listed before. |
| `Can_DisableControllerInterrupts()` `Can_EnableControllerInterrupts()` `Can_GetVersionInfo()` `Can_CheckWakeup()` `Can_GetStatus()` | Runtime has a constant limit. |
| `Can_Write()` | Runtime depends on the PDU size. |
| `Can_InterruptHandler()` | Runtime depends on the number of configured message objects, the interrupt/polling mode settings, the number of messages that have been sent since the last call, the number of messages that have been canceled since the last call, the number of messages that have been received since the last call, and the runtime of `CanIf` callback functions that correspond to each event listed before. |
| `Can_SetRxFilter()` | Runtime depends on the number of configured controllers and Number of HW Filter arrays. |
| `Can_Flush()` | Runtime depends on the number of queues stored in the TX buffer. |

## 5.10 Environment restrictions

The CAN driver's environment must consider the following restrictions:

- The CAN module interacts, among other modules (such as diagnostic event manager (DEM), default error tracer (DET), and Ecu state manager (ECUM)), with the CanIf module directly. The CAN driver never checks the actual origin of a request or the actual destination of a notification. The driver only sees the CanIf module as the origin and destination.
- Transmit cancellation may only be used when transmit buffers are enabled within the CanIf module.
- If a transmit is cancelled, the TX request for the new L-PDU must be repeated by the CanIf module within the notification function `CanIf_CancelTxConfirmation()`. For sequence-relevant streams, the sender must ensure that the next transmit request for the same CAN ID is only initiated after the last request was confirmed.
- If a request from the CanIf module conflicts with an event with a forced state transition (such as bus-off or wakeup), the request may be rejected depending on the timing (`CAN_E_TRANSITION` will be detected if DET is enabled). Because the CAN controller is in the STOPPED state, use `Can_SetControllerMode()` to change the state if necessary.
- In cases where multiple dedicated Tx buffers are configured with the same Message ID, the Tx buffers are not transmitted in the order of the Tx buffer number from lowest to highest because of hardware errata. If an application depends on the order of messages, you can request transmission for the next message after notifying the Tx confirmation of the previous message. This restriction occurs as well when using multiple Tx queue buffers configured with the same Message ID.
- According to the silicon errata, sporadic data corruption occurs during reception due to the following factors:
  - The number of active CAN channels in an instance
  - The data length code (DLC) of the received message

- The number of configured active filter element 11-bit IDs / 29-bit IDs.
- The arbitration bit rate
- The data bit rate

If the actual host clock frequency is lower than the minimum host clock frequency calculated from the above factors, sporadic data corruption occurs during data reception.

The following table shows the minimum host clock frequency for DLC = 5

Refer to the HW errata for the minimum host clock frequency with DLC setting in customer environments.

**Table 1. TRAVEO™ T2G: Minimum host clock frequency for CAN FD when DLC = 5**

| **Number of configured active filter element 11-bit IDs / 29-bit IDs** [1,2] | **Number of active CAN channels in an instance** | **Arbitration bit rate = 0.5 Mbps** | | | | **Arbitration bit rate = 1 Mbps** | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | **Data bit rate = 0.5 Mbps** | **Data bit rate = 1 Mbps** | **Data bit rate = 2 Mbps** | **Data bit rate = 4 Mbps** | **Data bit rate = 1 Mbps** | **Data bit rate = 2 Mbps** | **Data bit rate = 4 Mbps** | **Data bit rate = 5 Mbps** |
| 32 / 16 | 2 | 3.9 MHz | 7.1 MHz | 13.1 MHz | 22.8 MHz | 7.7 MHz | 14.1 MHz | 26.1 MHz | 31.5 MHz |
| | 3 | 5.4 MHz | 9.9 MHz | 18.3 MHz | 31.8 MHz | 10.7 MHz | 19.7 MHz | 36.5 MHz | 44.0 MHz |
| 64 / 32 | 2 | 7.4 MHz | 13.5 MHz | 24.9 MHz | 43.4 MHz | 14.7 MHz | 26.9 MHz | 49.8 MHz | 60.0 MHz |
| | 3 | 10.3 MHz | 18.8 MHz | 34.9 MHz | 60.7 MHz | 20.5 MHz | 37.6 MHz | 69.7 MHz | 84.0 MHz[4] |
| 96 / 48 | 2 | 10. 8 MHz | 19.9 MHz | 36.8 MHz | 64.0 MHz | 21.6 MHz | 39.7 MHz | 73.5 MHz | 88.6 MHz[4] |
| | 3 | 15.1 MHz | 27.8 MHz | 51.5 MHz | 89.6 MHz[4] | 30.2 MHz | 55.6 MHz | 102.9 MHz[3,4] | 124.0 MHz[3,4] |
| 128 / 64 | 2 | 14.3 MHz | 26.3 MHz | 48.6 MHz | 84.7 MHz[4] | 28.4 MHz | 52.5 MHz | 97.2 MHz[4] | 117.2 MHz[3,4] |
| | 3 | 20.0 MHz | 36.8 MHz | 68.0 MHz | 118.5 MHz[3,4] | 40.0 MHz | 73.5 MHz | 136.0 MHz[3,4] | 164.0 MHz[3] |

To avoid this data corruption, change the TRAVEO™ T2G MCAL CAN and MCU module configuration parameters, corresponding to each factor, as follows:

1) For the number of active CAN nodes: Use the maximum number of CanController configurations of a CAN instance that can be active (AUTOSAR controller state must be STARTED or SLEEP) at a time.
2) For the host clock frequency: In McuPeriGroupSettings, locate the setting with McuPeriGroup=MCU_PERI_GROUP5_MMIO5 and take the value from McuPeriGroupClockFrequency. For details, refer to the MCU module userguide. contact your Infineon sales representative for access to this document.
3) For the number of configured active filter element 11-bit IDs / 29-bit IDs”:

   Use the number of CanHwFilters in the CanHardwareObject with CanObjectType set to RECEIVE.

- 11-bit IDs: CanIdType set to STANDARD
- 29-bit IDs: CanIdType set to EXTENDED

4) For the Arbitration bit rate: Use the maximum CanControllerBaudRate value of all the CanControllers.
5) For the data bit rate: Use the maximum CanControllerFdBaudRate value of all the CanControllers if configured. Otherwise, use CanControllerBaudRate.

Also, use DLC ≥ 8 instead of DLCs 5, 6, and 7 in the CAN environment/system

## 5.11 Dynamically change Rx HW filter

The CAN HW filters of the Rx-dedicated buffer and Rx FIFO 0/1 can be changed dynamically using `Can_SetRxFilter()`. This can only be performed in the STOPPED state.

The implementation method for changing the filter is as follows:

- Use the `Can_RxFilterDataType` table and for the table structure, see 7.2.8 Can_RxFilterDataType. In the `Can_RxFilterDataType` table, define the hardware receive handle set in configuration and the associated filter information (set in `CanHwFilterCode` and `CanHwFilterMask`).
- Note that the possible ranges of `FilterIdValue` and `FilterMaskValue` vary depending on `CanIdType` (see Table 6). If they are out of range, an error will be triggered.

**Table 6 Range of FilterIdValue and FilterMaskValue**

| CanRxBufferSelection | CanIdType | RxFilter in the Can_RxFilterDataType | |
|---|---|---|---|
| | | FilterIdValue | FilterMaskValue |
| CAN_RX_DEDICATED | STANDARD | 0x0 to 0x7FF | 0x7FF |
| | EXTENDED | 0x0 to 0x1FFFFFFF | 0x1FFFFFFF |
| CAN_RX_FIFO0/1 | STANDARD | 0x0 to 0x7FF | 0x0 to 0x7FF |
| | EXTENDED | 0x0 to 0x1FFFFFFF | 0x0 to 0x1FFFFFFF |

- CAN_RX_FIFO0/1 can have multiple filters. However, the hardware receive handle values are the same for each filter. 0 is an example of the definition of CAN_RX_FIFO0/1. In this example, Rx FIFO 0 has two filters, as does Rx FIFO 1.

**Code listing 1 Example for RxFIFO0/1 multiple filters setting**

```
Can_RxFilterDataType RxFilters[] =
{
  {CanConf_(omit)_RxFIFO0_STD, {0x30u, 0x7F0u}},/*RxFIFO0*/\
  {CanConf_(omit)_RxFIFO0_STD, {0x40u, 0x7F0u}},/*RxFIFO0*/\
  {CanConf_(omit)_RxFIFO1_STD, {0x50u, 0x7F0u}},/*RxFIFO1*/\
  {CanConf_(omit)_RxFIFO1_STD, {0x60u, 0x7F0u}},/*RxFIFO1*/\
};
```

- Filters can basically be defined in any order. However, for hardware receive handles with multiple filters in Rx FIFO 0/1, they must be adjacent to each other (see 0). If you do not follow this rule, the incorrect filter will be set, and there will be no error detection for this incorrect filter.

**Code listing 2 Example for incorrect Rx FIFO 0 multiple filters setting**

```
Can_RxFilterDataType RxFilters[] =
{
  {CanConf_(omit)_RxFIFO0_STD, {0x30u, 0x7F0u}},/*RxFIFO0*/\
  {CanConf_(omit)_RxFIFO1_STD, {0x50u, 0x7F0u}},/*Incorrect*/\
  {CanConf_(omit)_RxFIFO0_STD, {0x40u, 0x7F0u}},/*RxFIFO0*/\
};
```

- `Can_SetRxFilter()` specifies the CAN controller in addition to the `Can_RxFilterDataType` table. If the `Can_RxFilterDataType` table contains a hardware receive handle associated with a CAN controller other than the specified one, changing the filter may take some time. Therefore, it is recommended to create a `Can_RxFilterDataType` table for each CAN controller (See 0).

**Code listing 3 Example for filters setting in one CAN controller**

```
Can_RxFilterDataType RxFilters_Ch0[] =
{
  {CanConf_(omit)_RxBuff_STD,   {0x10u, 0x7FFu}},\
  {CanConf_(omit)_RxBuff_EXT,   {0x20u, 0x1FFFFFFFu}},\
  {CanConf_(omit)_RxFIFO0_STD, {0x30u, 0x7F0u}},\
  {CanConf_(omit)_RxFIFO0_STD, {0x40u, 0x7F0u}},\
  {CanConf_(omit)_RxFIFO1_STD, {0x50u, 0x7F0u}},\
  {CanConf_(omit)_RxFIFO1_STD, {0x60u, 0x7F0u}},\
  {CanConf_(omit)_RxFIFO0_EXT, {0x70u, 0x1FFFFFF0u}},\
  {CanConf_(omit)_RxFIFO1_EXT, {0x80u, 0x1FFFFFF0u}},\
};
```

*Note:* It is recommended that *the hardware receive handle* use the *`CanHardwareObject symbolic name`* macro defined in *`Can_cfg.h`* (CanConf_CanHardwareObject_<Id of CanConfigSet>_Can_CanHardwareObject_<Name of CanHardwareObject>).

*Note: Filters that are not specified in this API will keep the settings that were initially registered with configuration. Therefore, it is recommended to set a filter value in the configuration to prevent the receiving of unintended data.*

*Note: The maximum number of filters for a given CAN controller is the number of filters for that CAN controller configured in the configuration. Therefore, the filter cannot be set beyond that maximum value.*

*Note: Undefined hardware receive handles will not change filter and will not detect errors.*

*Note: The filter for pretended networking mode cannot be changed.*

This feature is optional and can be enabled or disabled via the `CanSetRxFilterApi` configuration parameter.

## 5.12 Bus monitoring mode

Bus monitoring mode allows to receive the data frames, but does not transmit the messages and ACK because it only outputs recessive on the CAN bus. The mode can be switched using `Can_SetHwOperationMode()`. In addition to bus monitoring mode, normal mode in which transmission and reception is possible can be specified.

`Can_SetHwOperationMode()` can only be called when the CAN controller status is STOPPED or STARTED. By default, the CAN controller will be in normal mode. When the CAN controller changes state from STARTED to STOPPED, it returns to normal mode. It also returns to normal mode when the CAN controller state changes from STOPPED to SLEEP. After setting the CAN controller in bus monitoring mode, calling `Can_Write()` will return CAN_BUSY.

`Can_SetHwOperationMode()`, which switches between normal mode and bus monitoring mode, can be used by selecting the configuration parameter `CanSilentMode`.

Note: *When the CAN controller is in the STARTED state, switching from normal mode to bus monitoring mode causes the CAN controller to temporarily stop, which may have the following effects on transmission and reception:*

- *The data may get corrupted during the transmission. Therefore, do not switch to bus monitoring mode while transmitting.*
- *There is a possibility that you may lose the received data. Therefore, it is recommended that switching to bus monitoring mode is performed during a period where message loss is tolerated.*

Note: *If there are only the transmitting node and a node in bus monitoring mode, the transmitting node will detect the ACK error and continue to send the data. As a result, a node in bus monitor mode may receive the same message data many times, resulting in frequent receive interrupts. In this case, it is recommended to temporarily disable the interrupts using* `Can_DisableControllerInterrupts()`.

## 5.13 Stop transmission

The data stored in the TX queue can be cancelled. This feature stops the transmission of untransmitted data associated with the specified TxHandler. If transmission is cancelled, `CanIf_CancelTxconfirmation` is notified for the Tx buffers, but it is not notified for the Tx Queue and no transmission cancellation completion interrupt is generated. This feature uses `Can_Flush()` API.

This API can only be called when the CAN controller is in the STARTED state. This API is optional and can be enabled or disabled via the `CanFlushApi` configuration parameter.

## 5.14 Protocol error detection

This feature detects protocol errors and uses the protocol error call function to notify upper layer when the error state changes. Detectable errors are as follows:

- StuffError
- FormError
- AckError
- Bit1Error
- Bit0Error
- CRCError

The protocol error callout function notifies upper layer of protocol errors. However, these notifications occur only when there is a change in the error state, such as:

- Transition from Error Active to Error Passive
- Transition from Error Passive to Error Active
- Transition from Error Passive to Bus Off

This feature is optional and can be enabled or disabled via configuration parameter `CanProtocolError`. If the feature is enabled, the protocol error callout function can be used.

Note: *Protocol errors are not detected during the SLEEP state.*

## 5.15 Dynamically change any baud rate parameters

Any baud rate parameters can be changed dynamically. See 7.2.10 Can_BaudrateCustomType for the parameters that can be set. The setting range of each parameter is the same as `CanControllerBaudrateConfig` and `CanControllerFdBaudrateConfig` in the configuration (see 4.2.1.2 CanControllerBaudrateConfig and 4.2.1.3 CanControllerFdBaudrateConfig). Set `CstmFdEnabled=FALSE` if only Classic CAN is used. Set to 65535 (0xFFFF) if `CstmFdTrcvDelayCompensationOffset` is not used.

Verify that the `NominalPrescaler` is less than or equal to 512 by calculating `ClockFrequency`, `CstmTSeg1`, `CstmTSeg2`, and `CstmBaudrateValue`. If the `DataPrescaler` exceeds this upper limit, an error will be triggered.

$$\boxed{NominalPrescaler = \frac{ClockFrequency \times 1000}{(1+CstmTSeg1+CstmTSeg2)\times CstmBaudrateValue}}$$

Verify that the DataPrescaler is less than or equal to the upper limit for each condition by calculating ClockFrequency, CstmFdTSeg1, CstmFdTSeg2, and CstmFdBaudrateValue. If the `DataPrescaler` exceeds this upper limit, an error will be triggered.

$$\boxed{DataPrescaler = \frac{ClockFrequency \times 1000}{(1+CstmFdTSeg1+CstmFdTSeg2)\times CstmFdBaudrateValue}}$$

- Do not to use `FdTrcvDelayCompensationOffset` if DataPrescaler ≤ 32 (limitation by ISO-11898-1).
- Use CstmFdTrcvDelayCompensationOffset if DataPrescaler ≤ 2 (limitation by ISO-11898-1).

$$\boxed{CompensationOffset = \frac{ClockFrequency \times 1000\times CstmFdTrcvDelayCompensationOffset}{1000000}}$$

When using `CstmFdTrcvDelayCompensationOffset`, check if it is 400 or less.
If `CstmFdTrcvDelayCompensationOffset` exceeds 400, an error will be triggered.

This feature is optional and can be enabled or disabled via the `CanSetBaudrateAnyParametersApi` configuration parameter. If the function is enabled, any baudrate parameters can be changed dynamically using `Can_SetBaudrateAnyParameters()`.This API can only be called when the CAN controller is in STOPPED state.

The parameter information set by `Can_SetBaudrateAnyParameters()` is cleared by calling the existing baudrate setting API (`Can_SetBuadrate`, `Can_SetBaudrateInChangedClock`, `Can_ChangeBaudrate`).

*Note:* *Calling `Can_CheckBaudrate()` after calling `Can_SetBaudrateAnyParameters()` will result in an error.*

# 6 Hardware resources

## 6.1 Ports and pins

The TRAVEO™ T2G family features a different number of CAN controllers that comply with CAN specification Ver. 2.0, Part A and B. See section Hardware documentation.

To use CAN controllers, you must configure the RX and TX pins of the respective CAN controller within the PORT driver first.

Note: *The datasheet gives a list of all pins for all possible CAN controllers. Each derivative supports only a special subset of the pins and controllers.*

## 6.2 Timer

The CAN driver uses an OS timer for timeout supervision. A reference to the timer must be configured as `CanOsCounterRef;` the timeout value must be configured as `CanTimeoutDuration`.

The CAN driver does not use any hardware timers.

## 6.3 Interrupts

One interrupt (interrupt line 0) is used for each CAN instance. In some microcontrollers, only this line is available. This interrupt is shared with all interrupt sources of this instance (receive, transmit, error, and status interrupt). See the hardware events in 6.7 Hardware events.

The CAN driver provides the following interrupt service routines (category1 and category2) for each interrupt-configured CAN instance:

```
ISR_NATIVE(Can_Interrupt_<n>_Cat1)
ISR(Can_Interrupt_<n>_Cat2)
```

Note: *<n> refers to the physical CAN instance index, which means that if CANFD00 is configured to use interrupts, the interrupt service routines are named as follows:*

```
Can_Interrupt_CANFD00_Cat1()
Can_Interrupt_CANFD00_Cat2()
```

Note: *The OS must associate the named ISRs (in the generated files) with the corresponding CAN interrupt. For example, if CANFD00 is configured, then `Can_Interrupt_CANFD00_Cat2()` must be called from the OS interrupt service routine of CANFD00 (line 0) interrupt. In the case of category1 usage, the address of `Can_Interrupt_CANFD00_Cat1()` must be the entry for the CANFD00 interrupt in the OS interrupt vector table.*

Note: *On the Arm® Cortex®-M4 CPU, priority inversion of interrupts may occur under specific timing conditions in the integrated system with TRAVEO™ T2G MCAL. For more details, see the following errata notice.*

*Arm® Cortex®-M4 Software Developers Errata Notice - 838869:*
*"Store immediate overlapping exception return operation might vector to incorrect interrupt"*

*If the user application cannot tolerate the priority inversion, a DSB instruction should be added at the end of the interrupt function to avoid the priority inversion.*

*TRAVEO™ T2G MCAL interrupts are handled by an ISR wrapper (handler) in the integrated system. Thus, if necessary, the DSB instruction should be added just before the end of the handler by the integrator.*

## 6.4 CAN controller

Each configured Can controller is mapped to a physical CAN controller IP. The CAN driver exclusively controls all configured CAN controllers.

## 6.5 CAN message RAM

Each CAN controller is hard-wired to a CAN message RAM that it can read and write. The CAN message RAM is used for storing configuration lists and messages. The CAN driver stores the message body from the CAN message RAM in the stack area. In scope of the RX indication, the CAN driver provides pointers to the stack area where the message body is stored

The CAN message RAM is initialized by `Can_Init()` and `Can_DeInit()`.

## 6.6 Deep sleep mode

The wakeup functionality is not available while the MCU is in deep sleep mode due to the hardware architecture.

## 6.7 Hardware events

The CAN driver uses hardware events for each processing. Table 7 lists these events.

**Table 7 Hardware events**

| Processing | Hardware event | Description |
|---|---|---|
| `CanRxProcessing` | Message stored to Message ID Rx buffer | Received message is stored in a dedicated Rx buffer. |
| | Rx FIFO 0/1 new message | New message is written to Rx FIFO. |
| | Rx FIFO 0/1 message lost | Rx FIFO 0/1 message is lost. |
| | Protocol error | Protocol error detected. |
| | Error passive | Status changed to error passive. |
| `CanTxProcessing` | Tx event FIFO new entry | Tx handler wrote Tx event FIFO element. |
| | Transmission cancellation finished | Transmission cancellation is finished. |
| | Protocol error | Protocol error detected. |
| | Error passive | Status changed to error passive. |
| `CanBusoffProcessing` | Bus off status | Bus off status changed. |
| | ECC uncorrected bit error | Bit error detected, uncorrected. |
| | Protocol error | Protocol error detected. |

| | | |
|---|---|---|
| `CanWakeupProcessing` | Message stored to dedicated Rx buffer | Received message is stored in a dedicated Rx Buffer. |
| | Rx FIFO 0/1 new message | New message is written to Rx FIFO. |

# 7 Appendix A – API reference

## 7.1 Include files

If the CAN driver is used without the CAN interface, you must include the *Can.h* file within your application.

## 7.2 Data types

### 7.2.1 Can_PduType

**Type**

```
typedef struct can_pdutype
{
PduIdType swPduHandle;
uint8 length;
Can_IdType id;
uint8* sdu;
} Can_PduType;
```

**Description**

Provides the PDU information for the `Can_Write()` function.

### 7.2.2 Can_IdType

**Type**

```
typedef uint32 Can_IdType;
```

**Description**

Represents the identifier of the CAN ID.

### 7.2.3 Can_StateTransitionType

**Type**

```
typedef enum can_statetransitiontype
{
CAN_T_START,
CAN_T_STOP,
CAN_T_SLEEP,
CAN_T_WAKEUP
} Can_StateTransitionType;
```

**Description**

`Can_SetControllerMode()` uses these state transitions.

### 7.2.4 Can_ReturnType

**Type**

```
typedef enum can_returntype
{
CAN_OK,
CAN_NOT_OK,
CAN_BUSY
} Can_ReturnType;
```

**Description**

Returns the result of the operations of some CAN driver functions.

### 7.2.5 Can_HwHandleType

**Type**

```
typedef uint8 or uint16 Can_HwHandleType;
```

**Description**

Represents the hardware object handles of a CAN hardware unit.

In *Can_GeneralTypes.h* (Base module), `Can_HwHandleType` should be specify the following definition to the compiler.

`-DCAN_EXTENDED_HW_HANDLE` option is nothing: `Can_HwHandleType` type is `uint8`.
`-DCAN_EXTENDED_HW_HANDLE`: `Can_HwHandleType` type is `uint16`.

### 7.2.6 Can_HwType

**Type**

```
typedef struct can_hwtype_struct
{
Can_IdType CanId;
uint8 ControllerId;
Can_HwHandleType Hoh;
} Can_HwType;
```

**Description**

Data structure that provides a hardware object handle including its corresponding CAN controller and specific CAN ID.

### 7.2.7 Hardware-dependent data types

#### 7.2.7.1 Can_ConfigType

**Type**

Hardware-specific

**Description**

This is the type of the external data structure containing the overall initialization data of the CAN driver settings that affect all controllers.

#### 7.2.7.2 Can_ControllerConfigType

**Type**

Hardware-specific

**Description**

This is the type of the external data structure containing the overall initialization data for a single CAN controller.

#### 7.2.7.3 Can_ControllerBaudrateConfigType

**Type**

Hardware-specific

**Description**

This is the type of the external data structure containing the bit timing-related initialization data for a single CAN controller.

### 7.2.8 Can_RxFilterDataType

**Type**

```
typedef struct can_rxfilterdata_struct
{
Can_HwHandleType RxHwHandle;
Can_ControllerFilterType RxFilter;
} Can_RxFilterDataType;
```

**Description**

Provides the changing CAN Rx HW filter information for the `Can_SetRxFilter()` function.

### 7.2.9 Can_ControllerFilterType

**Type**

```
typedef struct can_controllerfiltertype_struct
{
uint32 FilterIdValue;
uint32 FilterMaskValue;
} Can_ControllerFilterType;
```

**Description**

This type stores the Can controller filter setting.

### 7.2.10 Can_BaudrateCustomType

**Type**

```
typedef struct can_baudratecustom_struct
{
uint16 CstmBaudrateValue;
uint8 CstmSyncJumpWidth;
uint8 CstmTSeg1;
uint8 CstmTSeg2;
boolean CstmFdEnabled;
uint8 CstmFdSyncJumpWidth;
uint8 CstmFdTSeg1;
uint8 CstmFdTSeg2;
boolean CstmFdTxBitRateSwitch;
uint16 CstmFdTrcvDelayCompensationOffset;
uint16 CstmFdBaudrateValue;
} Can_BaudrateCustomType;
```

**Description**

Provides the changing baud rate parameters for the `Can_SetBaudrateAnyParameters()` function.

## 7.3 Constants

### 7.3.1 Error codes

A service may return one error codes listed in Table 8 if default error detection is enabled.

**Table 8 Error codes**

| Name | Value | Description |
|---|---|---|
| `CAN_E_PARAM_POINTER` | 1 | Parameter is a NULL pointer. |
| `CAN_E_PARAM_HANDLE` | 2 | Parameter `Hth` is not a configured hardware transmit handle. |
| `CAN_E_PARAM_DLC` | 3 | Parameter `PduInfo->length` is greater than 8. |
| `CAN_E_PARAM_CONTROLLER` | 4 | Parameter `Controller` is out of range. |
| `CAN_E_UNINIT` | 5 | CAN driver is not yet initialized. |
| `CAN_E_TRANSITION` | 6 | The controller is not stopped. |
| `CAN_E_DATALOST` | 7 | Received CAN message is lost. |
| `CAN_E_PARAM_BAUDRATE` | 8 | Parameter `Baudrate` has an invalid value |
| `CAN_E_ICOM_CONFIG_INVALID` | 9 | Invalid ICOM configuration Id |
| `CAN_E_INIT_FAILED` | 10 | Invalid configuration set selection |

### 7.3.2 Vendor-specific error codes

In addition to the error codes given in section 7.3.1 Error codes, this CAN driver defines the errors listed in Table 9.

**Table 9 Vendor-specific error codes**

| Name | Value | Description |
|---|---|---|
| `CAN_E_PARAM_ID` | 11 | Parameter `Id` is out of range. |
| `CAN_E_UNKNOWN_DATA` | 12 | Received CAN message cannot be identified; invoked either from ISR or from the `Can_MainFunction_Read()` function. |
| `CAN_E_OS_TIME_REFUSED` | 13 | OS timer has refused its service. Timeout cannot be evaluated. |
| `CAN_E_UNCORRECTABLE_BIT_ERROR` | 14 | Uncorrectable bit error in message RAM is detected. |
| `CAN_E_PARAM_PDUID` | 15 | API service called with parameter `PduInfo->PduId` set to the value which is reserved for driver internal usage. Reserved value is the highest possible value of `PduId`. Depending on its type, this is either 0xFF (uint8) or 0xFFFF (uint16). |
| `CAN_E_IRQ_DEPTH` | 16 | An overflow for the call depth counter of the CAN driver interrupt functions happened. |

| Name | Value | Description |
|---|---|---|
| CAN_E_CALC_PRESCALER | 17 | The pre-scaler calculated by the changed clock is an unable to set.<br>Scope of nominal bit rate prescaler: 0x000-0x1FF<br>Scope of data bit rate prescaler: Transmitter delay compensation (`CanControllerTrcvDelayCompensationOffset`) is<br>Enabled: 0x01-0x02<br>Disabled: 0x01-0x20<br>(limitation by ISO-11898-1) |
| CAN_E_BAUDRATE_ANY_PARAM | 18 | API service called with wrong baudrate parameter. |
| CAN_E_PARAM_RXFILTER | 19 | API service called with wrong Rx filter parameter. |
| CAN_E_PARAM_MODE | 20 | API service called with wrong operation mode parameter. |

### 7.3.3 Version information

**Table 10 Version information**

| Name | Value | Description |
|---|---|---|
| CAN_AR_RELEASE_MAJOR_VERSION | 4 | Major version number (AUTOSAR). |
| CAN_AR_RELEASE_MINOR_VERSION | 2 | Minor version number (AUTOSAR). |
| CAN_AR_RELEASE_REVISION_VERSION | 2 | Patch version number (AUTOSAR). |
| CAN_SW_MAJOR_VERSION | see release notes | Vendor-specific major version number. |
| CAN_SW_MINOR_VERSION | see release notes | Vendor-specific minor version number. |
| CAN_SW_PATCH_VERSION | see release notes | Vendor-specific patch version number. |

### 7.3.4 Module information

**Table 11 Module information**

| Name | Value | Description |
|---|---|---|
| CAN_MODULE_ID | 80 | Module ID (Can) |
| CAN_VENDOR_ID | 66 | Vendor ID |

### 7.3.5 API service IDs

The API service IDs, listed in Table 12, are used when reporting errors via DET or via the error callout function.

**Table 12 API service IDs**

| Name | Value | API name |
|---|---|---|
| CAN_ID_INIT | 0x0 | Can_Init |
| CAN_ID_MF_WRITE | 0x1 | Can_MainFunction_Write |
| CAN_ID_SETCTRLMODE | 0x3 | Can_SetControllerMode |
| CAN_ID_DISABLECTRLINT | 0x4 | Can_DisableControllerInterrupts |
| CAN_ID_ENABLECTRLINT | 0x5 | Can_EnableControllerInterrupts |

| Name | Value | API name |
|---|---|---|
| CAN_ID_WRITE | 0x6 | Can_Write |
| CAN_ID_GETVERSIONINFO | 0x7 | Can_GetVersionInfo |
| CAN_ID_MF_READ | 0x8 | Can_MainFunction_Read |
| CAN_ID_MF_BUSOFF | 0x9 | Can_MainFunction_BusOff |
| CAN_ID_MF_WAKEUP | 0xA | Can_MainFunction_Wakeup |
| CAN_ID_CHECKWAKEUP | 0xB | Can_CheckWakeup |
| CAN_ID_MF_MODE | 0xC | Can_MainFunction_Mode |
| CAN_ID_CHANGEBAUDRATE | 0xD | Can_ChangeBaudrate |
| CAN_ID_CHECKBAUDRATE | 0xE | Can_CheckBaudrate |
| CAN_ID_SETBAUDRATE | 0xF | Can_SetBaudrate |
| CAN_ID_SETICOMCFG | 0xF | Can_SetIcomConfiguration |
| CAN_ID_DEINIT | 0x10 | Can_DeInit |
| CAN_ID_GETSTATUS | 0x20 | Can_GetStatus |
| CAN_ID_ISR | 0x21 | Can_InterruptHandler |
| CAN_ID_SETBAUDRATE_IN_CHANGED_CLOCK | 0x22 | Can_SetBaudrateInChangedClock |
| CAN_ID_SETRXFILTER | 0x23 | Can_SetRxFilter |
| CAN_ID_SETHWOPERATIONMODE | 0x24 | Can_SetHwOperationMode |
| CAN_ID_FLUSH | 0x25 | Can_Flush |
| CAN_ID_SETBAUDRATE_ANY_PARAMETERS | 0x26 | Can_SetBaudrateAnyParameters |

### 7.3.6 Can_GetStatus() bitmasks

The bitmasks listed in Table 13 are to be applied on the return value of function Can_GetStatus() to decode additional state information.

**Table 13 Can_GetStatus() bitmasks**

| Name | Value | Description |
|---|---|---|
| CAN_STATUS_ERROR_PASSIVE | 0x01 | CAN controller is in *error passive* state. |
| CAN_STATUS_RX_OVERFLOW | 0x10 | RX overflow has occurred since previous call of Can_GetStatus() |

## 7.4 Functions

### 7.4.1 Can_Init

**Syntax**

```
void Can_Init
(
const Can_ConfigType* Config
)
```

**Service ID**

0x00

**Sync/Async**

Synchronous

**Reentrancy**

Non-reentrant

**Parameters (in)**

- `Config` - Pointer to driver configuration.

**Parameters (out)**

None

**Return value**

None

**Errors**

- `CAN_E_INIT_FAILED` – Invalid configuration set selection.
- `CAN_E_PARAM_POINTER` – Either the `Config` parameter is a NULL pointer or configuration data is invalid.
- `CAN_E_TRANSITION` - The driver is not in 'uninitialized' state.

**Description**

CAN driver module initialization. The `Config` parameter is a pointer to the configuration that shall be used for initialization. All CAN controllers are in the `CAN_CS_STOPPED` state after initialization.

**Caveats**

This service must be called before any other service of the CAN driver is called.

### 7.4.2 Can_MainFunction_Write

**Syntax**

```
void Can_MainFunction_Write
(
void
)
```

**Service ID**

0x01

**Sync/Async**

Synchronous

**Reentrancy**

Non-reentrant

**Parameters (in)**

None

**Parameters (out)**

None

**Return value**

None

**Errors**

- `CAN_E_UNINIT` - Driver not initialized yet.

**Description**

This function performs the polling of TX confirmation and TX cancellation confirmation for each CAN controller that is configured for TX polling mode. (`CanTxProcessing` is set to `POLLING`).

**Caveats**

None

### 7.4.3 Can_SetControllerMode

**Syntax**

```
Can_ReturnType Can_SetControllerMode
(
uint8 Controller,
Can_StateTransitionType Transition
)
```

**Service ID**

0x03

**Sync/Async**

Asynchronous

**Reentrancy**

Reentrant for different controllers, non-reentrant for the same controller

**Parameters (in)**

- `Controller` - CAN controller for which the status shall be changed.
- `Transition` - A possible transition

**Parameters (out)**

None

**Return value**

- `CAN_OK` - Transition initiated.
- `CAN_NOT_OK` - Development or production error or wakeup during transition to 'sleep' mode occurred.

**Errors**

- `CAN_E_UNINIT` - Driver not initialized yet.
- `CAN_E_PARAM_CONTROLLER` - Parameter `Controller` is out of range.
- `CAN_E_TRANSITION` - The transition is not allowed. One or more CAN messages lost due to RX overflow.
- `CAN_E_OS_TIME_REFUSED` - OS timer has refused its service. Timeout cannot be evaluated.
- `CAN_E_IRQ_DEPTH` - Interrupt nesting violation occurred. Either a necessary interrupt lock could not be established, or it could not be released.

**Description**

This function performs a software-triggered state transition of the CAN controller state machine. This function enables necessary interrupts for the new state. It disables interrupts that are not allowed in the new state.

**Caveats**

The behavior of transmit operation is undefined while the software state is already `CAN_CS_STARTED`, but the CAN controller is not in operational mode yet. The upper layer must ensure that the previous call of `Can_SetControllerMode()` is returned before the function can be called again for the same controller.

`CAN_E_IRQ_DEPTH` can occur after the transition was successfully initiated. In that case, the return value is `E_OK`, although the error happened. Such an error is caused by another task that called `Can_EnableControllerInterrupts()` more often than `Can_DisableControllerInterrupts()`. The operation of `Can_SetControllerMode()` was successful, though.

`CAN_E_TRANSITION` can be generated when detecting a received message lost when transitioning from `CAN_CS_STARTED` to `CAN_CS_STOPPED`. In this case the transition fails.

### 7.4.4 Can_DisableControllerInterrupts

**Syntax**

```
void Can_DisableControllerInterrupts
(
uint8 Controller
)
```

**Service ID**

0x04

**Sync/Async**

Synchronous

**Reentrancy**

Reentrant

**Parameters (in)**

- `Controller` - CAN controller for which the interrupts shall be disabled.

**Parameters (out)**

None

**Return value**

None

**Errors**

- `CAN_E_UNINIT` - Driver not initialized yet.
- `CAN_E_PARAM_CONTROLLER` - Parameter `Controller` is out of range.
- `CAN_E_IRQ_DEPTH` - An overflow for the call depth counter of the CAN driver interrupt functions happened.

**Description**

This function disables all interrupts for this CAN controller. When `Can_DisableControllerInterrupts()` is called several times (without calling `Can_EnableControllerInterrupts()` in between) only the first call has any effect on the hardware. Further calls of `Can_DisableControllerInterrupts()` increase a counter that indicates how many `Can_ControllerEnableInterrupts` calls must be done in order to re-enable interrupts.

**Caveats**

None

### 7.4.5 Can_EnableControllerInterrupts

**Syntax**

```
void Can_EnableControllerInterrupts
(
uint8 Controller
)
```

**Service ID**

0x05

**Sync/Async**

Synchronous

**Reentrancy**

Reentrant

**Parameters (in)**

- `Controller` - CAN controller for which the interrupts shall be re-enabled.

**Parameters (out)**

None

**Return value**

None

**Errors**

- `CAN_E_UNINIT` - Driver not initialized yet.
- `CAN_E_PARAM_CONTROLLER` - Parameter `Controller` is out of range.
- `CAN_E_IRQ_DEPTH` - An overflow for the call depth counter of the CAN driver interrupt functions happened.

**Description**

This function enables all interrupts that must be enabled according to the current software status. When `Can_DisableControllerInterrupts()` has been called several times, `Can_EnableControllerInterrupts()` must be called as many times as `Can_DisableControllerInterrupts()` before the interrupts are re-enabled.

**Caveats**

None

### 7.4.6 Can_Write

**Syntax**

```
Can_ReturnType Can_Write
(
Can_HwHandleType Hth,
const Can_PduType* PduInfo
)
```

**Service ID**

0x06

**Sync/Async**

Synchronous

**Reentrancy**

Reentrant

**Parameters (in)**

- `Hth` - Information which HW-transmit handle shall be used for transmission.
- `PduInfo` - Pointer to SDU user memory, DLC, identifier, and SW PDU handle.

**Parameters (out)**

None

**Return value**

- `CAN_OK` - Write command has been accepted.
- `CAN_NOT_OK` - Development error occurred.
- `CAN_BUSY` - No TX hardware buffer available or preemptive call of `Can_Write()` that cannot be reentrant. Also call `Can_Write()` in pretended networking mode or bus monitoring mode.

**Errors**

- `CAN_E_UNINIT` - Driver not initialized yet.
- `CAN_E_PARAM_HANDLE` - Parameter `Hth` is not a configured hardware transmit handle.
- `CAN_E_PARAM_DLC` - `PduInfo->length` is invalid.
- `CAN_E_PARAM_POINTER` - Parameter `PduInfo` or `PduInfo->sdu` is a NULL pointer.
- `CAN_E_PARAM_ID` - CAN ID in `PduInfo->id` is out of range.
- `CAN_E_PARAM_PDUID` - `PduInfo->PduId` is set to the value which is reserved for driver internal usage. Reserved value is the highest possible value of `PduId`. Depending on its type, this is either 0xFF (uint8) or 0xFFFF (uint16).
- `CAN_E_TRANSITION` - The controller is not *started*.

**Description**

This function takes the pointer `PduInfo` and sends the data using the hardware transmit handle `Hth`.

**Caveats**

None

### 7.4.7 Can_GetVersionInfo

**Syntax**

```
void Can_GetVersionInfo
(
Std_VersionInfoType *versioninfo
)
```

**Service ID**

0x07

**Sync/Async**

Synchronous

**Reentrancy**

Reentrant

**Parameters (in)**

None

**Parameters (out)**

- `versioninfo` - Pointer to where to store the version information of this module.

**Return value**

None

**Errors**

- `CAN_E_PARAM_POINTER` - `versioninfo` is a NULL pointer.

**Description**

This function returns the version information of this module.

**Caveats**

None

### 7.4.8 Can_MainFunction_Read

**Syntax**

```
void Can_MainFunction_Read
(
void
)
```

**Service ID**

0x08

**Sync/Async**

Synchronous

**Reentrancy**

Non-reentrant

**Parameters (in)**

None

**Parameters (out)**

None

**Return value**

None

**Errors**

- `CAN_E_UNINIT` - Driver not initialized yet.
- `CAN_E_UNKNOWN_DATA` - Received CAN message cannot be identified.
- `CAN_E_DATALOST` - One or more CAN messages lost due to RX overflow.
- `CAN_E_OS_TIME_REFUSED` - OS timer has refused its service. Timeout cannot be evaluated.

**Description**

This function performs the polling of RX indications for each CAN controller that is configured for RX polling mode. (`CanRxProcessing` is set to `POLLING`).

**Caveats**

`CAN_E_OS_TIME_REFUSED` can occur when `CanIcomMissingMessageTimerValue` is enabled and cannot evaluate the timeout in pretended networking mode.

### 7.4.9 Can_MainFunction_BusOff

**Syntax**

```
void Can_MainFunction_BusOff
(
void
)
```

**Service ID**

0x09

**Sync/Async**

Synchronous

**Reentrancy**

Non-reentrant

**Parameters (in)**

None

**Parameters (out)**

None

**Return value**

None

**Errors**

- `CAN_E_UNINIT` - Driver not initialized yet.
- `CAN_E_UNCORRECTABLE_BIT_ERROR` - An uncorrectable bit error was detected.

**Description**

This function performs the polling of bus-off and ECC error events that are configured statically as 'to be polled'.

**Caveats**

None

### 7.4.10 Can_MainFunction_Wakeup

**Syntax**

```
void Can_MainFunction_Wakeup
(
void
)
```

**Service ID**

0x0A

**Sync/Async**

Synchronous

**Reentrancy**

Non-reentrant

**Parameters (in)**

None

**Parameters (out)**

None

**Return value**

None

**Errors**

- `CAN_E_UNINIT` - Driver not initialized yet.

**Description**

This function performs the polling of wake-up events that are configured statically as 'to be polled'.

**Caveats**

None

### 7.4.11 Can_CheckWakeup

**Syntax**

```
Can_ReturnType Can_CheckWakeup
(
uint8 Controller
)
```

**Service ID**

0x0B

**Sync/Async**

Synchronous

**Reentrancy**

Non-reentrant

**Parameters (in)**

- `Controller` - CAN controller to be checked for wakeup.

**Parameters (out)**

None

**Return value**

- `E_OK` - A wakeup was detected for the given controller.
- `E_NOT_OK` - No wakeup was detected for the given controller.

**Errors**

- `CAN_E_UNINIT` - Driver not initialized yet.
- `CAN_E_PARAM_CONTROLLER` - Parameter `Controller` is out of range.

**Description**

This function checks if a wakeup has occurred for the given controller.

**Caveats**

None

### 7.4.12 Can_MainFunction_Mode

**Syntax**

```
void Can_MainFunction_Mode
(
void
)
```

**Service ID**

0x0C

**Sync/Async**

Synchronous

**Reentrancy**

Non-reentrant

**Parameters (in)**

None

**Parameters (out)**

None

**Return value**

None

**Errors**

- `CAN_E_UNINIT` - Driver not initialized yet.
- `CAN_E_OS_TIME_REFUSED` - OS timer has refused its service. Timeout cannot be evaluated.
- `CAN_E_IRQ_DEPTH` - Interrupt nesting violation occurred. Either a necessary interrupt lock could not be established, or it could not be released.

**Description**

This function performs the polling of CAN controller mode transitions.

**Caveats**

`CAN_E_IRQ_DEPTH` can occur after the transition was successfully initiated. Such an error is caused by another task that called `Can_EnableControllerInterrupts()` more often than `Can_DisableControllerInterrupts()`.

### 7.4.13 Can_ChangeBaudrate

**Syntax**

```
Std_ReturnType Can_ChangeBaudrate
(
uint8 Controller,
uint16 Baudrate
)
```

**Service ID**

0x0D

**Sync/Async**

Synchronous

**Reentrancy**

Reentrant for different controllers, non-reentrant for the same controller

**Parameters (in)**

- `Controller` - CAN controller whose baud rate shall be changed.
- `Baudrate` - Requested baud rate in kbps.

**Parameters (out)**

None

**Return value**

- `E_OK` - Service request accepted, baud rate change started.
- `E_NOT_OK` - Service request not accepted.

**Errors**

- `CAN_E_UNINIT` - Driver not initialized yet.
- `CAN_E_PARAM_CONTROLLER` - Parameter `Controller` is out of range.
- `CAN_E_PARAM_BAUDRATE` - Parameter `Baudrate` has an invalid value.
- `CAN_E_TRANSITION` - Controller is not stopped.

**Description**

This function changes the baud rate of the CAN controller.

**Caveats**

This function re-initializes the CAN controller and the controller-specific settings.

### 7.4.14 Can_CheckBaudrate

**Syntax**

```
Std_ReturnType Can_CheckBaudrate
(
uint8 Controller,
uint16 Baudrate
)
```

**Service ID**

0x0E

**Sync/Async**

Synchronous

**Reentrancy**

Reentrant

**Parameters (in)**

- `Controller` - CAN controller to be checked for baud rate support.
- `Baudrate` - Baud rate value (in kbps) to be checked.

**Parameters (out)**

None

**Return value**

- `E_OK` - Baud rate supported by the CAN controller.
- `E_NOT_OK` - Baud rate not supported / invalid CAN controller.

**Errors**

- `CAN_E_UNINIT` - Driver not initialized yet.
- `CAN_E_PARAM_CONTROLLER` - Parameter `Controller` is out of range.
- `CAN_E_PARAM_BAUDRATE` - Parameter `Baudrate` has an invalid value. The baud rate parameters must be set using `Can_SetBaudrateAnyParameters` before calling `Can_CheckBaudrate`.

**Description**

This function checks if a certain CAN controller supports a requested baud rate.

**Caveats**

The call context shall be on task level (polling mode).

The CAN must be initialized after power on.

### 7.4.15 Can_SetBaudrate

**Syntax**

```
Std_ReturnType Can_SetBaudrate
(
uint8 Controller,
uint16 BaudRateConfigID
)
```

**Service ID**

0x0F

**Sync/Async**

Synchronous

**Reentrancy**

Reentrant for different controllers, non-reentrant for the same controller

**Parameters (in)**

- `Controller` - CAN controller whose baud rate shall be changed.
- `BaudRateConfigID` - References a baud rate configuration by ID (see `CanControllerBaudRateConfigID`).

**Parameters (out)**

None

**Return value**

- `E_OK` - Service request accepted, setting of (new) baud rate started.
- `E_NOT_OK` - Service request not accepted.

**Errors**

- `CAN_E_UNINIT` - Driver not initialized yet.
- `CAN_E_PARAM_CONTROLLER` - Parameter `Controller` is out of range.
- `CAN_E_PARAM_BAUDRATE` - Parameter `BaudRateConfigID` has an invalid value.
- `CAN_E_TRANSITION` - Controller is not stopped.

**Description**

This function changes the baud rate configuration of the CAN controller according to parameter `BaudRateConfigID`.

**Caveats**

This function re-initializes the CAN controller and the controller-specific settings.

### 7.4.16 Can_SetBaudrateInChangedClock

**Syntax**

```
Std_ReturnType Can_SetBaudrateInChangedClock
(
uint8 Controller,
uint16 BaudRateConfigID,
Can_ClkFrequencyType ClockFrequency
)
```

**Service ID**

0x22

**Sync/Async**

Synchronous

**Reentrancy**

Reentrant for different controllers, non-reentrant for the same controller

**Parameters (in)**

- `Controller` - CAN controller whose baud rate shall be changed.
- `BaudRateConfigID` - References a baud rate configuration by ID (see `CanControllerBaudRateConfigID`).
- `ClockFrequency` - Changed clock in MHz.

**Parameters (out)**

None

**Return value**

- `E_OK` - Service request accepted, setting of (new) baud rate started.
- `E_NOT_OK` - Service request not accepted.

**Errors**

- `CAN_E_UNINIT` - Driver not initialized yet.
- `CAN_E_PARAM_CONTROLLER` - Parameter `Controller` is out of range.
- `CAN_E_PARAM_BAUDRATE` - Parameter `BaudRateConfigID` has an invalid value.
- `CAN_E_TRANSITION` - Controller is not stopped.
- `CAN_E_CALC_PRESCALER` - The pre-scaler calculated by the changed clock is an unable to set.

**Description**

This function is an extension to the AUTOSAR CAN driver specification.

This function changes the baud rate configuration of the CAN controller in changed clock according to parameter `BaudRateConfigID`.

**Caveats**

This function re-initializes the CAN controller and the controller-specific settings.

### 7.4.17 Can_SetIcomConfiguration

**Syntax**

```
Std_ReturnType Can_SetIcomConfiguration
(
uint8 Controller,
IcomConfigIdType ConfigurationId
)
```

**Service ID**

0xF

**Sync/Async**

Synchronous

**Reentrancy**

Reentrant for different controllers, non-reentrant for the same controller

**Parameters (in)**

- `Controller` - CAN controller for which the status shall be changed
- `ConfigurationId` - Requested configuration

**Parameters (out)**

None

**Return value**

- `E_OK` – CAN driver succeeded in setting a configuration with a valid configuration id
- `E_NOT_OK` - CAN driver failed to set a configuration with a valid configuration id

**Errors**

- `CAN_E_UNINIT` - Driver not initialized yet.
- `CAN_E_PARAM_CONTROLLER` - Parameter `Controller` is out of range.
- `CAN_E_ICOM_CONFIG_INVALID` - Invalid ICOM configuration id.
- `CAN_E_TRANSITION` - Controller is not started.
- `CAN_E_OS_TIME_REFUSED` – OS timer has refused its service. Timeout cannot be evaluated.

**Description**

This function shall change the ICOM configuration of a CAN controller to the requested one.

**Caveats**

`ConfigurationId` = 0 unconditionally invalidates pretended networking mode.

### 7.4.18 Can_GetStatus

**Syntax**

```
uint8 Can_GetStatus
(
uint8 Controller
)
```

**Service ID**

0x20

**Sync/Async**

Synchronous

**Reentrancy**

Reentrant

**Parameters (in)**

- `Controller` - CAN controller for which the status will be retrieved.

**Parameters (out)**

None

**Return value**

Any (bit ORed) combination of following flags:

- `CAN_STATUS_ERROR_PASSIVE` - CAN controller is in error passive state.
- `CAN_STATUS_RX_OVERFLOW` - At least one RX overflow has occurred since the previous calling of `Can_GetStatus()`.

If the return value is 0x00, the status is undefined (for example, the conditions above have not occurred, or the CAN controller is not in the `STARTED` state).

**Errors**

- `CAN_E_UNINIT` - Driver not initialized yet.
- `CAN_E_PARAM_CONTROLLER` - Parameter `Controller` is out of range.

**Description**

This service is an extension to the AUTOSAR CAN driver specification. It returns some additional state information which may be required by the application.

- `CAN_STATUS_ERROR_PASSIVE` - This flag is returned if the error counters of the CAN macro indicate that the controller is in the error passive state. The status is updated on every `Can_GetStatus()` call.
- CAN_STATUS_RX_OVERFLOW - The update of the internal state for `Can_GetStatus()` happens every time new data is received by this controller and is processed by the corresponding function (`Can_MainFunction_Read()` call in case of polling mode configuration or on every CAN RX interrupt in case of IRQ mode configuration). This flag is only returned once, if no new RX overflow condition occurs during subsequent calls to `Can_GetStatus()`.

**Caveats**

The bus-off notification specified by AUTOSAR takes precedence over error passive information. This means that the function may return `CAN_STATUS_ERROR_PASSIVE` while the controller is in bus-off state.

### 7.4.19 Can_DeInit

**Syntax**

```
void Can_DeInit
(
void
)
```

**Service ID**

0x10

**Sync/Async**

Synchronous

**Reentrancy**

Non-reentrant

**Parameters (in)**

None

**Parameters (out)**

None

**Return value**

None

**Errors**

- `CAN_E_TRANSITION` – CAN driver is not initialized yet, or the controller has started.

**Description**

De-initializes the module.

**Caveats**

To call this function, all CAN controllers must be in the Stop or Sleep state.

### 7.4.20 Can_SetRxFilter

**Syntax**

```
void Can_SetRxFilter
(
uint8 Controller,
const Can_RxFilterDataType *RxFilterDataPtr,
uint16 NumOfRxFilterData
)
```

**Service ID**

0x23

**Sync/Async**

Synchronous

**Reentrancy**

Reentrant for different Controllers. Non-reentrant for same Controller.

**Parameters (in)**

- `Controller` - CAN controller for which the CanHwFilter shall be rewrite
- `RxFilterDataPtr` - Pointer to `Can_RxFilterDataType` table that store values for rewriting CanHwFilter
- `NumOfRxFilterData` - Number of Can_RxFilterDataType table

**Parameters (out)**

None

**Return value**

- `CAN_OK` - Service request accepted, setting of new filter
- `CAN_NOT_OK` - Service request not accepted

**Errors**

- `CAN_E_UNINIT` - Driver not initialized yet
- `CAN_E_PARAM_CONTROLLER` - Parameter `Controller` is out of range
- `CAN_E_PARAM_POINTER` - `RxFilterDataPtr` is a NULL pointer
- `CAN_E_TRANSITION` - Controller is not stopped
- `CAN_E_PARAM_RXFILTER` - `RxFilterDataPtr-> RxFilter.FilterIdValue` or `RxFilterDataPtr-> RxFilter.FilterMaskValue` exceeds standard or extended value

**Description**

This function changes Rx filter dynamically.

**Caveats**

The HW filter settings are stored in ROM (Flash) on configuration, but are copied to RAM for rewriting.

### 7.4.21 Can_SetHwOperationMode

**Syntax**

```
void Can_SetHwOperationMode
(
uint8 Controller,
uint8 Mode
)
```

**Service ID**

0x24

**Sync/Async**

Synchronous

**Reentrancy**

Reentrant

**Parameters (in)**

- `Controller` - CAN controller for which the operation mode shall be changed
- `Mode` - Requested operation mode. (CAN_T_NORMAL/CAN_T_MONITORING)

**Parameters (out)**

None

**Return value**

- `CAN_OK` - Service request accepted, setting of operation mode
- `CAN_NOT_OK` - Service request not accepted

**Errors**

- `CAN_E_UNINIT` - Driver not initialized yet
- `CAN_E_PARAM_CONTROLLER` - Parameter `Controller` is out of range
- `CAN_E_PARAM_MODE` - Parameter `Mode` is wrong operation mode
- `CAN_E_TRANSITION` - Controller is not started or stopped

**Description**

This function changes the operation mode of a CAN controller.

**Caveats**

None

### 7.4.22 Can_Flush

**Syntax**

```
void Can_Flush
(
Can_HwHandleType Hth
)
```

**Service ID**

0x25

**Sync/Async**

Synchronous

**Reentrancy**

Reentrant

**Parameters (in)**

- `Hth` - Information which HW-transmit handle shall be used for transmit cancel

**Parameters (out)**

None

**Return value**

- `CAN_OK` - Service request accepted
- `CAN_NOT_OK` - Service request not accepted

**Errors**

- `CAN_E_UNINIT` - Driver not initialized yet
- `CAN_E_PARAM_HANDLE` - Parameter `Hth` is not a configured hardware transmit handle
- `CAN_E_TRANSITION` - Controller is not started

**Description**

This function cancels the transmission of unsent data stored in the TX queue and TX buffer.

**Caveats**

Even if the transmission data is not cancelled in time, the return value will be CAN_OK.

### 7.4.23 Can_SetBaudrateAnyParameters

**Syntax**

```
void Can_SetBaudrateAnyParameters
(
uint8 Controller,
const Can_BaudrateCustomType *BaudrateParamPtr,
Can_ClkFrequencyType ClockFrequency
)
```

**Service ID**

0x26

**Sync/Async**

Synchronous

**Reentrancy**

Reentrant for different Controllers. Non-reentrant for same Controller.

**Parameters (in)**

- `Controller` - CanController to change baudrate parameter
- `BaudrateParamPtr` - Various parameters for setting buadrate
- `ClockFrequency` - Clock supplied to CanController[MHz]

**Parameters (out)**

None

**Return value**

- `E_OK` - Service request accepted, setting of (new) baud rate started
- `E_NOT_OK` - Service request not accepted

**Errors**

- `CAN_E_UNINIT` - Driver not initialized yet
- `CAN_E_PARAM_CONTROLLER` - API Service called with wrong parameter: controller index
- `CAN_E_PARAM_POINTER` - `BaudrateParamPtr` is a NULL pointer
- `CAN_E_TRANSITION` - Controller is not stopped
- `CAN_E_BAUDRATE_ANY_PARAM` - API service called with wrong parameters used to calculate the data prescaler
- `CAN_E_CALC_PRESCALER` - The prescaler calculated by the changed clock is unable to set. See 5.15 Dynamically change any baud rate parameters for the formula.

**Description**

This function changes various baud rate parameters dynamically.

**Caveats**

This function re-initializes the CAN controller and the controller-specific settings.

## 7.5 Required callback functions

### 7.5.1 CAN interface

The following callback functions provided by the CAN interface are used by the CAN driver. If you do not use the CAN interface, you must implement these functions within your application.

#### 7.5.1.1 CanIf_TxConfirmation

**Syntax**

```
void CanIf_TxConfirmation
(
PduIdType CanTxPduId
)
```

**Sync/Async**

Synchronous

**Reentrancy**

Reentrant

**Parameters (in)**

- `CanTxPduId` - L-PDU handle of CAN L-PDU successfully transmitted.

**Return value**

None

**Description**

Indicates a successful transmission. It is either called by the TX interrupt service routine of the corresponding HW resource or within the `Can_MainFunction_Write()` in case of polling mode.

#### 7.5.1.2 CanIf_RxIndication (Compliant to AUTOSAR 4.0.3)

**Syntax**

```
void CanIf_RxIndication
(
Can_HwHandleType Hrh,
Can_IdType CanId,
uint8 CanDlc,
const uint8* CanSduPtr
)
```

**Sync/Async**

Synchronous

**Reentrancy**

Reentrant

**Parameters (in)**

- `Hrh` - Hardware receives handle.
- `CanId` - Identifier of the received CAN message.
- `CanDlc` - DLC of the received CAN message.
- `CanSduPtr` - Pointer to the received L-SDU (payload).

**Return value**

None

**Description**

Indicates that a new CAN message was received. It is either called by the RX interrupt service routine of the corresponding HW resource or within the `Can_MainFunction_Read()` in case of polling mode.

`CanIf_RxIndication()` is called using this signature, if `CanRxIndicationCompatibility` is configured to `CAN_ASR_403_COMPATIBILITY`.

#### 7.5.1.3 CanIf_RxIndication (compliant to AUTOSAR 4.2.2)

**Syntax**

```
void CanIf_RxIndication
(
const Can_HwType* Mailbox,
const PduInfoType* PduInfoPtr
)
```

**Sync/Async**

Synchronous

**Reentrancy**

Reentrant

**Parameters (in)**

- `Mailbox` - Identifies the HRH and its corresponding CAN controller.
- `PduInfoPtr` - Pointer to the received L-PDU.

**Return value**

None

**Description**

Indicates that a new CAN message was received. It is either called by the RX interrupt service routine of the corresponding HW resource or within `Can_MainFunction_Read()` in case of polling mode.

`CanIf_RxIndication()` is called using this signature, if `CanRxIndicationCompatibility` is configured to `CAN_ASR_421_COMPATIBILITY`.

### 7.5.1.4 CanIf_CancelTxConfirmation

**Syntax**

```
void CanIf_CancelTxConfirmation
(
PduIdType CanTxPduId,
const PduInfoType* PduInfoPtr
)
```

**Sync/Async**

Synchronous

**Reentrancy**

Non-reentrant

**Parameters (in)**

- `CanTxPduId` - L-PDU handle of CAN L-PDU that was canceled.
- `PduInfoPtr` - Pointer to the `PduInfo` struct of the L_PDU that was canceled. `PduInfo` will be freed after the return from `CanIf_CancelTxConfirmation()`.

**Return value**

None

**Description**

Indicates a cancellation of a transmission. It is either called by the TX interrupt service routine of the corresponding HW resource or within `Can_MainFunction_Write()` in case of polling mode.

*Note:* *When a message is canceled, the buffer will remain blocked until `CanIf_CancelTxConfirmation()` is called because `PduInfo` needs to be provided. This means that in polling mode, the buffer will remain blocked at least until the next cycle of `Can_MainFunction`, regardless of whether cancellation is enabled.*

*Note:* *This callback function has been removed in AUTOSAR 4.2.2, but this CAN driver supports the function.*

### 7.5.1.5 CanIf_ControllerBusOff

**Syntax**

```
void CanIf_ControllerBusOff
(
uint8 Controller
)
```

**Sync/Async**

Synchronous

**Reentrancy**

Reentrant

**Parameters (in)**

- `Controller` - Specifies the CAN controller that detected bus-off.

**Return value**

None

**Description**

Indicates that the controller switched to the bus-off state. It is called by the bus-off interrupt service routine of the corresponding controller or by `Can_MainFunction_BusOff()` if the bus-off event is polled.

### 7.5.1.6 CanIf_ControllerModeIndication

**Syntax**

```
void CanIf_ControllerModeIndication
(
uint8 Controller,
CanIf_ControllerModeType ControllerMode
)
```

**Sync/Async**

Synchronous

**Reentrancy**

Reentrant

**Parameters (in)**

- `Controller` - Specifies the CAN controller whose state has changed.
- `ControllerMode` - Specifies the mode to which the CAN controller has changed.

**Return value**

None

#### Description

Indicates a state transition of the corresponding CAN controller. It is called from within `Can_SetControllerMode()` or `Can_MainFunction_Mode()`.

### 7.5.1.7 CanIf_TriggerTransmit

#### Syntax

```
Std_ReturnType CanIf_TriggerTransmit
(
PduIdType TxPduId,
PduInfoType* PduInfoPtr
)
```

#### Sync/Async

Synchronous

#### Reentrancy

Reentrant for different PduIds, non-reentrant for the same PduId

#### Parameters (in)

- `TxPduId` - ID of the SDU that is requested to be transmitted.
- `PduInfoPtr` - Contains a pointer to a buffer (`SduDataPtr`) where the SDU data must be copied to. On return, the service indicates the length of the copied SDU data in `SduLength`.

#### Return value

- `E_OK` - SDU has been copied and `SduLength` indicates the number of copied bytes.
- `E_NOT_OK` - No SDU data has been copied. `PduInfoPtr` must not be used because it may contain a NULL pointer or point to invalid data.

#### Description

Within this API, `CanIf` must copy its data into the buffer provided by `PduInfoPtr->SduDataPtr` and update the length of the actual copied data in `PduInfoPtr->SduLength`.

`CanIf_TriggerTransmit()` is called only if `CanTriggerTransmitEnable` is enabled in the configuration, no errors are existent, and the pointer to SDU user memory that is passed to Can_Write is NULL.

#### 7.5.1.8 CanIf_CurrentIcomConfiguration

**Syntax**

```
void CanIf_CurrentIcomConfiguration
(
uint8 Controller,
IcomConfigIdType ConfigurationId,
IcomSwitch_ErrorType Error
)
```

**Sync/Async**

Synchronous

**Reentrancy**

Reentrant for different controllers, non-reentrant for the same controller

**Parameters (in)**

- `Controller` - Specifies the CAN controller that enabled the pretended networking mode.
- `ConfigurationId` - Active configuration Id.
- `Error - ICOM_SWITCH_E_OK`: No error
- `ICOM_SWITCH_E_FAILED`: Switch to the requested configuration failed. This is a severe error.

**Return value**

None

**Description**

Informs about the change of the ICOM configuration of a CAN controller using the abstract `CanIf` controller ID.

### 7.5.2 DET

If default error detection is enabled, the CAN driver uses the following callback function provided by DET. If you do not use DET, you must implement this function within your application.

#### 7.5.2.1 Det_ReportError

**Syntax**

```
Std_ReturnType Det_ReportError
(
uint16 ModuleId,
uint8 InstanceId,
uint8 ApiId,
uint8 ErrorId
)
```

**Reentrancy**

Reentrant

**Parameters (in)**

- `ModuleId` - Module ID of calling module.
- `InstanceId` - Instance ID of calling module.
- `ApiId` - ID of the API service that calls this function.
- `ErrorId` - ID of the detected development error.

**Return value**

Returns always `E_OK`.

**Description**

Service for reporting development errors.

### 7.5.3 DEM

If DEM notifications are enabled, the CAN driver uses the following callback function provided by DEM. If you do not use DEM, you must implement this function within your application.

#### 7.5.3.1 Dem_ReportErrorStatus

**Syntax**

```
void Dem_ReportErrorStatus
(
Dem_EventIdType EventId,
Dem_EventStatusType EventStatus
)
```

**Reentrancy**

Reentrant

**Parameters (in)**

- `EventId` - Event ID assigned by the DEM module.
- `EventStatus` - Monitor test result of a given event.

**Return value**

None

**Description**

Service for reporting diagnostic events.

### 7.5.4 AUTOSAR OS

The following functions provided by AUTOSAR OS are used by the CAN driver. If you do not use AUTOSAR OS, you must implement these functions within your application.

#### 7.5.4.1 GetCounterValue

**Syntax**

```
StatusType GetCounterValue
(
CounterType CounterID,
TickRefType Value
)
```

**Sync/Async**

Synchronous

**Reentrancy**

Reentrant

**Parameters (in)**

- `CounterID` - Specifies the counter whose tick value is to be read.

**Parameters (out)**

- `Value` - Contains the current tick value of the counter.

**Return value**

- `E_OK` - No errors.
- `E_OS_ID` - Invalid parameter `CounterID`.

**Description**

Reads the current count value of the counter specified by the `CounterID` parameter for which the CAN driver uses the timer reference configured in `CanOsCounterRef`.

#### 7.5.4.2 GetElapsedValue

**Syntax**

```
StatusType GetElapsedValue
(
CounterType CounterID,
TickRefType Value,
TickRefType ElapsedValue
)
```

**Sync/Async**

Synchronous

**Reentrancy**

Reentrant

**Parameters (in)**

- `CounterID` - Specifies the counter to be read.

**Parameters (inout)**

- `Value` – `In`: The previously read tick value of the counter. `Out`: The current tick value of the counter.

**Parameters (out)**

- `ElapsedValue` - The difference between the value currently read and the value previously read.

**Return value**

- `E_OK` - No errors.
- `E_OS_ID` - Invalid parameter `CounterID`.
- `E_OS_VALUE` - Invalid parameter `Value`.

**Description**

Calculates the number of ticks between the current tick value and a previously read tick value of the counter specified by the `CounterID` parameter for which the CAN driver uses the timer reference configured in `CanOsCounterRef`.

### 7.5.5 Callout functions

#### 7.5.5.1 L-PDU callout API

The AUTOSAR CAN module supports optional L-PDU callouts on every reception of an L-PDU, in spite of `CanIf_RxIndication()`. The name of the L-PDU callout function is configured by `CanLPduReceiveCalloutFunction`.

**Syntax**

```
boolean LPDU_CalloutName
(
Can_HwHandleType Hrh,
Can_IdType CanId,
uint8 CanDlc,
const uint8 *CanSduPtr
)
```

**Sync/Async**

Synchronous

**Reentrancy**

Reentrant

**Parameters (in)**

- `Hrh` - Hardware receive handle.

- `CanId` - Received CAN frame identifier.
- `CanDlc` - Received CAN frame length.
- `CanSduPtr` - Received CAN frame data pointer.

**Return value**

- `TRUE` - The L-PDU shall be processed.
- `FALSE` - The L-PDU shall not be processed any further.

**Description**

`LPDU_CalloutName()` must be substituted with the concrete L-PDU callout name, which is configurable. This function uses `Can_HwHandleType` as an argument. Therefore, include *Can_GeneralTypes.h* (Base module).

#### 7.5.5.2 Error callout API

The AUTOSAR CAN module requires an error callout handler. Each error is reported to this handler; error checking cannot be switched OFF. The name of the function to be called can be configured by the `CanErrorCalloutFunction` parameter.

**Syntax**

```
void Error_Handler_Name
(
uint16 ModuleId,
uint8 InstanceId,
uint8 ApiId,
uint8 ErrorId
)
```

**Reentrancy**

Reentrant

**Parameters (in)**

- `ModuleId` - Module ID of calling module.
- `InstanceId` - Instance ID of calling module.
- `ApiId` - ID of the API service that calls this function.
- `ErrorId` - ID of the detected error.

**Return value**

None

**Description**

Service for reporting errors.

### 7.5.5.3 Protocol error callout API

Protocol error callout API is called when the error state changes (see 5.14 Protocol error detection) and notifies upper layer of various protocol errors, status, and Tx/Rx error counts. This feature is available when the `CanProtocolError` parameter is enabled. This parameter can be enabled or disabled each CAN controller. The name of the function to be called can be configured by the `CanProtocolErrorCalloutFunction` parameter.

It is either called by the interrupt service routine of the corresponding HW resource or within MainFunction in case of polling mode.

**Syntax**

```
void ProtocolErrorCalloutHandler
(
uint8 Controller,
uint8 Status,
uint8 TxErrorCnt,
uint8 RxErrorCnt,
uint8 ErrorReason
)
```

**Reentrancy**

Reentrant

**Parameters (in)**

- `Controller` - CAN controller
- `Status` - ErrorActive(=0)/ErorrPassive(=1)/BusOff(=2)
- `TxErrorCnt` - Tx Error count
- `RxErrorCnt` - Rx Error count
- `ErrorReason` - StuffError(=1)/FormError(=2)/AckError(=3)/Bit1Error(=4)/Bit0Error(=5)/CRCError(=6)

**Return value**

None

**Description**

To declare this function and use the entity, you must prepare the header and source file.

# 8 Appendix B – Access register table

## 8.1 TT_CANFD

Table 14 TT_CANFD access register table

| Register | Bit No. | Access size | Value | Description | Timing | Mask value | Monitoring value |
|---|---|---|---|---|---|---|---|
| ECC_CTL (Set in each instance) | 31:0 | Word (32 bits) | 0x00010000 | Enable ECC for CAN-FD SRAM. | Can_Init | 0x00010000 | 0x00010000 |
| | | | 0x00000000 | Disable ECC for CAN-FD SRAM. | Can_DeInit | 0x00010000 | 0x00000000 |
| RXFTOP_CTL | 31:0 | Word (32 bits) | 0x00000003 | FIFO0 and FIFO1 top pointer enable. | Can_Init | 0x00000003 | 0x00000003 |
| | | | 0x00000000 | FIFO0 and FIFO1 top pointer disable. | Can_DeInit | 0x00000003 | 0x00000000 |
| RXTOP0_DATA | 31:0 | Word (32 bits) | - | Receive FIFO0 top data | Received messages | 0x00000000 (Monitoring is not needed.) | 0x00000000 (Monitoring is not needed.) |
| RXTOP1_DATA | 31:0 | Word (32 bits) | - | Receive FIFO1 top data | Received messages | 0x00000000 (Monitoring is not needed.) | 0x00000000 (Monitoring is not needed.) |

| Register | Bit No. | Access size | Value | Description | Timing | Mask value | Monitoring value |
|---|---|---|---|---|---|---|---|
| `DBTP` | 31:0 | Word (32 bits) | 0x00000000 \| (TransmitterDelayCompensation<<23) \| (DataBitRatePrescaler<<16) \| (DataTSeg1<<8) \| (DataTSeg2<<4) \| (DataSyncJumpWidth) | Set data bit timing & prescaler in CAN-FD buadrate configuration | `Can_SetControllerMode` (State transition : STOP to START / STOP to SLEEP) | 0x009F1FFF | 0x00****** (* Depend on configuration value or calculated value obtained by Can_SetBaudrateInChangedClock) |
| | | | 0x00000A33 | Clear data bit timing & prescaler in CAN-FD buadrate configuration | `Can_DeInit` | 0x009F1FFF | 0x00000A33 |
| `RWD` | 31:0 | Word (32 bits) | 0x00000000 | Set RAM watchdog (This register is disabled because 0 is set) | `Can_SetControllerMode` (State transition : STOP to START / STOP to SLEEP) `Can_DeInit` | 0x0000FFFF | 0x00000000 |

| Register | Bit No. | Access size | Value | Description | Timing | Mask value | Monitoring value |
|---|---|---|---|---|---|---|---|
| `CCCR` | 31:0 | Word (32 bits) | 0x00000003 | Stop CAN controller | `Can_Init`<br>`Can_SetControllerMode`<br>(State transition :<br>STOP to STOP /<br>START to STOP /<br>SLEEP to STOP)<br>Occurred Bus-Off<br>Occurred uncorrectable bit error<br>Receive wakeup massage | 0x00000003 | 0x00000003 |
| | | | 0x00000000 \| (BitRateSwitchEnable<<9) \| (FDOperationEnable<<8) | Start CAN controller | `Can_SetControllerMode`<br>(State transition :<br>STOP to START) | 0x00000323 | 0x00000*00<br>(* Depend on configuration value) |
| | | | 0x00000020 \| (BitRateSwitchEnable<<9) \| (FDOperationEnable<<8) | Sleep CAN controller | `Can_SetControllerMode`<br>(State transition :<br>STOP to SLEEP) | 0x00000323 | 0x00000*20<br>(* Depend on configuration value) |
| | | | 0x00000001 | Initialize CAN controller | `Can_DeInit` | 0xFFFFFFFF | 0x00000001 |
| | | | 0x00000020 \| (BitRateSwitchEnable<<9) \| (FDOperationEnable<<8) | Bus monitoring mode | `Can_SetHwOperationMode`<br>(State transition :<br>START, Mode: MONITORING) | 0x00000323 | 0x00000*20<br>(* Depend on configuration value) |

| Register | Bit No. | Access size | Value | Description | Timing | Mask value | Monitoring value |
|---|---|---|---|---|---|---|---|
| NBTP | 31:0 | Word (32 bits) | 0x00000000 \| (NominalSyncJump Width<<25) \| (NominalBitRatePre scaler<<16) \| (NominalTSeg1<<8) \| (NominalTSeg2) | Set nominal bit timing & prescaler in CAN baudrate configuration | `Can_SetControllerMode` (State transition : STOP to START / STOP to SLEEP) | 0xFFFFFFFF | 0x******** (*Depend on configuration value or calculated value obtained by Can_SetBaudrateInChangedClock) |
| | | | 0x06000A03 | Clear nominal bit timing & prescaler in CAN baudrate configuration | `Can_DeInit` | 0xFFFFFFFF | 0x06000A03 |
| TSCC | 31:0 | Word (32 bits) | 0x00000000 | Set timestamp counter configuration (This register is disabled because 0 is set) | `Can_SetControllerMode` (State transition : STOP to START / STOP to SLEEP) `Can_DeInit` | 0x000F0003 | 0x00000000 |
| TOCC | 31:0 | Word (32 bits) | 0x00000000 | Set timeout counter configuration (This register is disabled because 0 is set) | `Can_SetControllerMode` (State transition : STOP to START / STOP to SLEEP) | 0xFFFF0007 | 0x00000000 |
| | | | 0xFFFF0000 | Clear timeout counter configuration | `Can_DeInit` | 0xFFFF0007 | 0xFFFF0000 |
| ECR | 31:0 | Word (32bits) | - | Get transmit error counter and receive error passive | This register is read only and is not related to timing. | 0x00000000 (Monitoring is not needed.) | 0x00000000 (Monitoring is not needed.) |

| Register | Bit No. | Access size | Value | Description | Timing | Mask value | Monitoring value |
|---|---|---|---|---|---|---|---|
| `PSR` | 31:0 | Word (32 bits) | - | Get Bus-off status<br>Get last error code in pretended networking mode (Stuff error / Form error / Ack error / Bit 1 error / Bit 0 error / CRC error) | This register is read only and is not related to timing. | 0x00000000<br>(Monitoring is not needed.) | 0x00000000<br>(Monitoring is not needed.) |
| `TDCR` | 31:0 | Word (32 bits) | 0x00000000 \| (TransmitterDelayCompensationOffset <<8) | Set transmitter delay compensation offset of CAN-FD baudrate configuration | `Can_SetControllerMode`<br>(State transition :<br>STOP to START /<br>STOP to SLEEP) | 0x00007F00 | 0x0000**00<br>(* Depend on configuration value) |
|  |  |  | 0x00000000 | Clear transmitter delay compensation offset of CAN-FD baudrate configuration | `Can_DeInit` | 0x00007F00 | 0x00000000 |

| Register | Bit No. | Access size | Value | Description | Timing | Mask value | Monitoring value |
|---|---|---|---|---|---|---|---|
| `IR` | 31:0 | Word (32 bits) | 0x00000000 \| (InterrptRegisterFlag) | Get interrupt cause and clear | `Can_MainFunction_BusOff` `Can_MainFunction_Read` `Can_MainFunction_Write` `Can_MainFunction_Wakeup` `Can_SetControllerMode` (State transition : START to STOP) Bus-Off interrupt Received interrupt Transmission finished interrupt Transmission cancelled interrupt Rx FIFO0/1 message lost Bit error uncorrected Protocol error interrupt | 0x00000000 (Monitoring is not needed.) | 0x00000000 (Monitoring is not needed.) |
| | | | 0x00000000 | Clear interrupt cause | `Can_SetControllerMode` (State transition : STOP to START / STOP to SLEEP) Can_DeInit | 0xFFFFFFFF | 0x00000000 |
| `IE` | 31:0 | Word (32 bits) | 0x00000000 \| (BusOffInterruptEnable <<25) \| (BitErrorUncorrectedInterruptEnable<<21) \| (DedicatedRxBufferInterruptEnable<<19) \| (TxEventFIFONewEntryInterruptEnable<<12) \| | Set interrupt enable | `Can_SetControllerMode` (State transition : STOP to START / STOP to SLEEP) | 0x02281499 | 0x0******* (* Depend on configuration value) |

| Register | Bit No. | Access size | Value | Description | Timing | Mask value | Monitoring value |
|---|---|---|---|---|---|---|---|
| | | | (TransmissionCancellationFinishedInterruptEnable<<10) \| (RxFIFO1MessageLostInterruptEnable<<7) \| (RxFIFO1NewMessageInterruptEnable<<4) \| (RxFIFO0MessageLostInterruptEnable<<3) \| (RxFIFO0NewMessageInterruptEnable) | | | | |
| | | | 0x00000000 | Clear interrupt enable | `Can_Init` `Can_MainFunction_BusOff` `Can_MainFunction_Wakeup` `Can_SetControllerMode` (State transition : STOP to STOP / START to STOP / SLEEP to STOP) Can_DeInit | 0x02281499 | 0x00000000 |
| | | | 0x00000000 \| (BusOffInterruptEnable <<25) \| (BitErrorUncorrectedInterruptEnable<<21) \| (DedicatedRxBufferInterruptEnable<<19) \| | Set protocol error interrupt enable/disable | `Can_SetControllerMode` (State transition : STOP to START) | 0x1AA81499 | 0x******** (* Depend on configuration value) |

| Register | Bit No. | Access size | Value | Description | Timing | Mask value | Monitoring value |
|---|---|---|---|---|---|---|---|
| | | | (TxEventFIFONewEntryInterruptEnable<<12) \| (TransmissionCancellationFinishedInterruptEnable<<10) \| (RxFIFO1MessageLostInterruptEnable<<7) \| (RxFIFO1NewMessageInterruptEnable<<4) \| (RxFIFO0MessageLostInterruptEnable<<3) \| (RxFIFO0NewMessageInterruptEnable) \| (ProtocolErrorInDataPhaseEnable <<28) \| (ProtocolErrorInArbitrationPhaseEnable <<27) \| (ErrorPassiveInterruptEnable<<23) | | | | |
| | | | 0x00000000 \| (ProtocolErrorInDataPhaseEnable <<28) \| (ProtocolErrorInArbitrationPhaseEnable <<27) | Set protocol error frame interrupt enable/disable | `Can_SetIcomConfiguration` (Pretended networking mode transitions from disable to enable under the condition of `CanIcomPayloadLengthError`=TRUE | 0xFFFFFFFF | 0x0******* (* Depend on configuration value) |

| Register | Bit No. | Access size | Value | Description | Timing | Mask value | Monitoring value |
|---|---|---|---|---|---|---|---|
| | | | | | Pretended networking mode transitions from enable to disable) | | |
| `ILS` | 31:0 | Word (32 bits) | 0x00000000 \| (BusOffInterruptLine <<25) \| (BitErrorUncorrectedInterruptLine<<21) \| (DedicatedRxBufferInterruptLine<<19) \| (TxEventFIFONewEntryInterruptLine<<12) \| (TransmissionCancellationFinishedInterruptLine<<10) \| (RxFIFO1MessageLostInterruptLine<<7) \| (RxFIFO1NewMessageInterruptLine<<4) \| (RxFIFO0MessageLostInterruptLine<<3) \| (RxFIFO0NewMessageInterruptLine) | Set interrupt line select (Set to int 0 if corresponding interrupt is enabled in IE, otherwise set to int 1.) | `Can_SetControllerMode` (State transition : STOP to START / STOP to SLEEP) | 0x02281499 | 0x0******* (* Depend on configuration value) |
| | | | 0x00000000 \| (BusOffInterruptEnable <<25) \| (BitErrorUncorrectedInterruptEnable<<21) \| | Set protocol error interrupt enable or disable | `Can_SetControllerMode` (State transition : STOP to START) | 0x1AA81499 | 0x******** (* Depend on configuration value) |

| Register | Bit No. | Access size | Value | Description | Timing | Mask value | Monitoring value |
|---|---|---|---|---|---|---|---|
| | | | (DedicatedRxBufferInterruptEnable<<19) \| (TxEventFIFONewEntryInterruptEnable<<12) \| (TransmissionCancellationFinishedInterruptEnable<<10) \| (RxFIFO1MessageLostInterruptEnable<<7) \| (RxFIFO1NewMessageInterruptEnable<<4) \| (RxFIFO0MessageLostInterruptEnable<<3) \| (RxFIFO0NewMessageInterruptEnable) \| (ProtocolErrorInDataPhaseEnable <<28) \| (ProtocolErrorInArbitrationPhaseEnable <<27) \| (ErrorPassiveInterruptEnable<<23) | | | | |
| | | | 0x00000000 | Clear interrupt line select | `Can_DeInit` | 0xFFFFFFFF | 0x00000000 |

| Register | Bit No. | Access size | Value | Description | Timing | Mask value | Monitoring value |
|---|---|---|---|---|---|---|---|
| | | | 0x00000000 \| (ProtocolErrorInDataPhaseLine <<28) \| (ProtocolErrorInArbitrationPhaseLine <<27) | Set protocol error frame interrupt line select int0/int1 | `Can_SetIcomConfiguration` (Pretended networking mode transitions from disable to enable under the condition of `CanIcomPayloadLengthError`=TRUE Pretended networking mode transitions from enable to disable) | 0xFFFFFFFF | 0x0******* (* Depend on configuration value) |
| `ILE` | 31:0 | Word (32 bits) | 0x00000001 | Set interrupt line enable (EINT0 use only) | `Can_EnableControllerInterrupts` `Can_MainFunction_Mode` `Can_SetControllerMode` (State transition : STOP to START / STOP to SLEEP In the case where interrupt is enabled) | 0x00000003 | 0x00000001 |
| | | | 0x00000000 | Clear interrupt line enable | `Can_DisableControllerInterrupts` `Can_MainFunction_Mode` `Can_SetControllerMode` (State transition : STOP to START / STOP to SLEEP In the case where interrupt is disabled) Can_DeInit | 0x00000003 | 0x00000000 |
| `GFC` | 31:0 | Word (32 bits) | 0x0000003F | Set global filter configuration | `Can_Init` `Can_SetRxFilter` | 0x0000003F | 0x0000003F |
| | | | 0x00000000 | Clear global filter configuration | `Can_DeInit` | 0x0000003F | 0x00000000 |

| Register | Bit No. | Access size | Value | Description | Timing | Mask value | Monitoring value |
|---|---|---|---|---|---|---|---|
| SIDFC | 31:0 | Word (32 bits) | 0x00000000 \| (StandardIDFilterListSize<<16) \| (StandardIDFilterStartAddress<<2) | Set standard ID filter configuration | `Can_Init` `Can_SetRxFilter` | 0x00FFFFFC | 0x00****** (* Depend on configuration value) |
| | | | 0x00000000 | Clear standard ID filter configuration | `Can_DeInit` | 0x00FFFFFC | 0x00000000 |
| XIDFC | 31:0 | Word (32 bits) | 0x00000000 \| (ExtendedIDFilterListSize<<16) \| )ExtendedIDFilterStartAddress<<2) | Set extended ID filter configuration | `Can_Init` `Can_SetRxFilter` | 0x007FFFFC | 0x00****** (* Depend on configuration value) |
| | | | 0x00000000 | Clear extended ID filter configuration | `Can_DeInit` | 0x007FFFFC | 0x00000000 |
| XIDAM | 31:0 | Word (32 bits) | 0x1FFFFFFF | Set extended ID AND Mask (Mask is not active) | `Can_Init` `Can_SetRxFilter` `Can_DeInit` | 0x1FFFFFFF | 0x1FFFFFFF |
| NDAT1 | 31:0 | Word (32 bits) | - | Get new Data1 and clear (Rx buffer 0 to 31) | Received messages | 0x00000000 (Monitoring is not needed.) | 0x00000000 (Monitoring is not needed.) |
| | | | 0x00000000 | Clear new Data1 (Rx buffer 0 to 31) | `Can_DeInit` | 0xFFFFFFFF | 0x00000000 |
| NDAT2 | 31:0 | Word (32 bits) | - | Get new Data2 and clear (Rx buffer 0 to 31) | When CAN controller is starting. When message is received and CAN controller wakes up. | 0x00000000 (Monitoring is not needed.) | 0x00000000 (Monitoring is not needed.) |
| | | | 0x00000000 | Clear new Data1 (Rx buffer 0 to 31) | `Can_DeInit` | 0xFFFFFFFF | 0x00000000 |

| Register | Bit No. | Access size | Value | Description | Timing | Mask value | Monitoring value |
|---|---|---|---|---|---|---|---|
| RXF0C | 31:0 | Word (32 bits) | 0x81000000 \| (RxFIFO0Size<<16) \| (RxFIFO0StartAddress<<2) | Set Rx FIFO0 configuration (Rx FIFO0 operation is blocking mode. Rx FIFO0 watermark interrupt is level 1.) | Can_Init Can_SetRxFilter | 0xFF7FFFFC | 0x81****** (* Depend on configuration value) |
| | | | 0x00000000 | Clear Rx FIFO0 configuration | Can_DeInit | 0xFF7FFFFC | 0x00000000 |
| RXF0S | 31:0 | Word (32 bits) | - | Get Rx FIFO0 status. | Received messages | 0x00000000 (Monitoring is not needed.) | 0x00000000 (Monitoring is not needed.) |
| RXF0A | 31:0 | Word (32 bits) | - | Get FIFO0 acknowledge index and clear | Received messages | 0x00000000 (Monitoring is not needed.) | 0x00000000 (Monitoring is not needed.) |
| | | | 0x00000000 | Clear FIFO0 acknowledge index | Can_DeInit | 0x0000003F | 0x00000000 |
| RXBC | 31:0 | Word (32 bits) | 0x00000000 \| (RxBufferStartAddress<<2) | Set Rx buffer configuration | Can_Init Can_SetRxFilter | 0x0000FFFC | 0x0000**** (* Depend on configuration value) |
| | | | 0x00000000 | Clear Rx buffer configuration | Can_DeInit | 0x0000FFFC | 0x00000000 |
| RXF1C | 31:0 | Word (32 bits) | 0x81000000 \| (RxFIFO1Size<<16) \| (RxFIFO1StartAddress<<2) | Set Rx FIFO1 configuration (Rx FIFO1 operation is blocking mode. Rx FIFO1 watermark interrupt is level 1.) | Can_Init Can_SetRxFilter | 0xFF7FFFFC | 0x81****** (* Depend on configuration value) |
| | | | 0x00000000 | Clear Rx FIFO1 configuration | Can_DeInit | 0xFF7FFFFC | 0x00000000 |

| Register | Bit No. | Access size | Value | Description | Timing | Mask value | Monitoring value |
|---|---|---|---|---|---|---|---|
| `RXF1S` | 31:0 | Word (32 bits) | - | Get Rx FIFO1 status. | Received messages | 0x00000000 (Monitoring is not needed.) | 0x00000000 (Monitoring is not needed.) |
| `RXF1A` | 31:0 | Word (32 bits) | - | Get FIFO1 acknowledge index and clear | Received messages | 0x00000000 (Monitoring is not needed.) | 0x00000000 (Monitoring is not needed.) |
| | | | 0x00000000 | Clear FIFO1 acknowledge index | `Can_DeInit` | 0x0000003F | 0x00000000 |
| `RXESC` | 31:0 | Word (32 bits) | 0x00000000 \| (RxBufferDataFieldSize<<8) \| (RxFIFO1DataFieldSize<<4) \| (RxFIFO0DataFieldSize) | Set Rx buffer / FIFO element size configuration | `Can_Init` `Can_SetRxFilter` | 0x00000777 | 0x00000*** (* Depend on configuration value) |
| | | | 0x00000000 | Clear Rx buffer / FIFO element size configuration | `Can_DeInit` | 0x00000777 | 0x00000000 |
| `TXBC` | 31:0 | Word (32 bits) | 0x00000000 \| (TransmitQueueSize<<24) \| (NumberOfDedicatedTransmitBuffers<<16) \| (TxBufferStartAddress<<2) | Set Tx buffer configuration (Queue mode operation only) | `Can_Init` | 0x7F3FFFFC | 0x******** (* Depend on configuration value, however Tx operation is queue mode only(bit[30]=1)) |
| | | | 0x00000000 | Clear Tx buffer configuration | `Can_DeInit` | 0x7F3FFFFC | 0x00000000 |
| `TXFQS` | 31:0 | Word (32 bits) | - | Get Tx queue put index 0 to 31 | `Can_Write` | 0x00000000 (Monitoring is not needed.) | 0x00000000 (Monitoring is not needed.) |

| Register | Bit No. | Access size | Value | Description | Timing | Mask value | Monitoring value |
|---|---|---|---|---|---|---|---|
| `TXESC` | 31:0 | Word (32 bits) | 0x00000000 \| (TxBufferDataFieldSize) | Set Tx buffer element size configuration | `Can_Init` | 0x00000003 | 0x0000000* (* Depend on configuration value) |
| | | | 0x00000000 | Clear Tx buffer element size configuration | `Can_DeInit` | 0x00000003 | 0x00000000 |
| `TXBAR` | 31:0 | Word (32 bits) | - | Set Tx buffer Add request | `Can_Write` | 0x00000000 (Monitoring is not needed.) | 0x00000000 (Monitoring is not needed.) |
| `TXBCR` | 31:0 | Word (32 bits) | - | Set Tx buffer cancellation request | `Can_Write` `Can_Flush` `Can_SetControllerMode` (Execute transmission cancellation if transmission data remains when transitioning from START to STOP) | 0x00000000 (Monitoring is not needed.) | 0x00000000 (Monitoring is not needed.) |
| `TXBTO` | 31:0 | Word (32 bits) | - | Get Tx buffer transmission occurred (Used with TXBCF to check if transmission cancellation is occurring) | transmitssion cancellation occured | 0x00000000 (Monitoring is not needed.) | 0x00000000 (Monitoring is not needed.) |
| `TXBCF` | 31:0 | Word (32 bits) | - | Get Tx buffer cancellation finished (Used with TXBTO to check if transmission cancellation is occurring) | transmitssion cancellation occured | 0x00000000 (Monitoring is not needed.) | 0x00000000 (Monitoring is not needed.) |

| Register | Bit No. | Access size | Value | Description | Timing | Mask value | Monitoring value |
|---|---|---|---|---|---|---|---|
| TXBTIE | 31:0 | Word (32 bits) | 0x00000000 \| (TxBufferInterruptEnable) | Set Tx buffer transmission interrupt enable (Transmission interrupts are set to enable for the number of Tx handlers in CanHardwareObject regardless of "INTERRUPT" or "POLLING" in CanTxProcessing) | `Can_Init` | 0xFFFFFFFF | 0x******** (* Depend on configuration value) |
| | | | 0x00000000 | Clear Tx buffer transmission interrupt enable | `Can_DeInit` | 0xFFFFFFFF | 0x00000000 |
| TXBCIE | 31:0 | Word (32 bits) | 0x00000000 \| (TxBufferCancellationFinishedInterruptEnable) | Set Tx buffer cancellation finished interrupt enable (Cancellation interrupts are set to enable for the number of Tx handlers in CanHardwareObject regardless of "INTERRUPT" or "POLLING" in CanTxProcessing) | `Can_Init` | 0xFFFFFFFF | 0x******** (* Depend on configuration value) |
| | | | 0x00000000 | Clear Tx buffer cancellation finished interrupt enable | `Can_DeInit` | 0xFFFFFFFF | 0x00000000 |

| Register | Bit No. | Access size | Value | Description | Timing | Mask value | Monitoring value |
|---|---|---|---|---|---|---|---|
| TXEFC | 31:0 | Word (32 bits) | 0x01000000 \| (TxEventFIFOSize<< 16) \| (TxEventFIFOStartAddress<<2) | Set Tx event FIFO configuration (Event FIFO watermark interrupt is level 1.) | `Can_Init` | 0x3F3FFFFC | 0x01****** (* Depend on configuration value) |
| | | | 0x00000000 | Clear Tx event FIFO configuration | `Can_DeInit` | 0x3F3FFFFC | 0x00000000 |
| TXEFS | 31:0 | Word (32 bits) | - | Get Tx event FIFO index and fill level | transmitssion cancellation occured | 0x00000000 (Monitoring is not needed.) | 0x00000000 (Monitoring is not needed.) |
| TXEFA | 31:0 | Word (32 bits) | - | Set Tx event FIFO acknowledge | `Can_MainFunction_Write` Transmitssion finished interrupt `Can_SetControllerMode` (Clear transmission cancellation if transmission data remains when transitioning from START to STOP) | 0x00000000 (Monitoring is not needed.) | 0x00000000 (Monitoring is not needed.) |
| | | | 0x00000000 | Clear Tx event FIFO acknowledge | `Can_DeInit` | 0x0000001F | 0x00000000 |

## Revision history

| Document revision | Date | Description of changes |
|---|---|---|
| ** | 2018-06-28 | New spec. |
| *A | 2018-12-17 | Hardware Documentation<br>Added two TRAVEO™ T2G Automotive Body Controller High Family TRMs<br>Deleted the datasheet<br>4.2.1 CanController<br>Deleted CanFilterMask and CanFilterMaskValue.<br>Added sentence to the Annotation field of CanControllerBaudRate<br>4.2.2 CanHardwareObject<br>Deleted CanFilterMaskRef.<br>Changed of sentences in Description field in CanHwFilterMask.<br>4.2.3 CanIcom<br>Added sentence to the Annotation field of CanIcomWakeOnBusOff CanIcomSignalMaskH/L, CanIcomSignalValueH/L and CanIcomSignalRef.<br>Changed of sentences in Range field of CanIcomMessageId and CanIcomMessageIdMask<br>5.1.5 Reception (RX) Filter Parameters<br>Deleted the words CanFilterMaskValue and CanFilterMaskRef.<br>A.2.5 Can_HwHandleType<br>Changed the type of Can_HwHandleType to support both uint8 and uint16.<br>A.5.5 Callout functions<br>Changed the argument Hrh of LPDU_CalloutName from uint8 to Can_HwHandleType. |
| *B | 2019-06-11 | Hardware Documentation<br>Updated hardware documentation information.<br>4.3 Implementation Constants<br>Deteled the description of “precompile time".<br>5.2 Initialization<br>Changed CanController macro name from “CanConf_CanController_<name of CanController>” to “CanConf_CanController_<Id of CanConfigSet>_<name of CanController>” specified in Can_SetControllerMode function.<br>A.3.2 Vendor Specific Error Code<br>Changed the scope of Data Bit Rate Prescale in CAN_E_CALC_PRESCALER.<br>B.1.1 TTD<br>IE: Added setting for calling Can_SetIcomConfiguration<br>ILS: Added setting for calling Can_SetIcomConfiguration |
| *C | 2020-09-05 | 2.6 Memory Mapping<br>Changed Can_MemMap.h file include folder |
| *D | 2020-11-19 | MOVED TO INFINEON TEMPLATE. |
| *E | 2021-05-17 | Added 6.6 Deep sleep mode. |

| Document revision | Date | Description of changes |
|---|---|---|
| *F | 2021-08-27 | 5.1.2 CAN controller state machine<br>Changed Figure 5 CAN controller state machine<br>5.1.2.5 State transitions<br>Added description of ECC error detection in Sleep mode<br>5.10 Environment restrictions<br>Added behavior associated with event contention<br>Added a note in 6.3 Interrupts<br>7.4.12 Can_MainFunction_Mode<br>Added description about DET error to items "Errors" and "Caveats"<br>Table 14 TT_CANFD access register table<br>Updated ILE register |
| *G | 2021-09-07 | 2.6.1 Memory allocation keyword<br>Moved "Pointer to whole configuration setting" from "`SEC_CONST_ASIL_B_UNSPECIFIED`" to "`SEC_VAR_INIT_ASIL_B_UNSPECIFIED`" |
| *H | 2021-12-07 | Updated to the latest branding guidelines |
| *I | 2023-03-23 | 2.6.2 Assembler code<br>Added new chapter<br>4.2.1.3.6 CanControllerTrcvDelayCompensationOffset<br>Added detailed information with a calculation formula<br>6.3 Interrupts<br>Added description of reference to hardware events<br>6.7 Hardware events<br>Added new chapter<br>4.2.1.1.1 CanBusoffProcessing<br>7.4.9 Can_MainFunction_BusOff<br>Added description of ECC error event |
| *J | 2023-06-01 | 5.10 Environment restrictions<br>Added restriction associated with message order |
| *K | 2023-10-06 | 5.10 Environment restrictions<br>Added restriction associated with sporadic data corruption during reception. |
| *L | 2023-12-08 | Web release. No content updates. |
| *M | 2024-07-30 | Updated 2.6.1 Memory allocation keyword<br>Updated 3.1 Static files<br>Added 3.4.10 Protocol error callout handler<br>Added 4.1.14 CanSetRxFilterApi<br>Added 4.1.15 CanSetBaudrateAnyParametersApi<br>Added 4.1.16 CanSilentMode<br>Added 4.1.17 CanFlushApi<br>Added 4.1.27 CanProtocolErrorCalloutFunction<br>Added 4.2.1.1.14 CanProtocolError |

| Document revision | Date | Description of changes |
|---|---|---|
| | | Updated 5.4 Runtime baud rate change<br>5.5 API parameter checking<br>Added new error code to Table 3 Development error codes<br>5.9 Execution time dependencies<br>Added new affected function to Table 5 Execution time dependencies<br>Added 5.11 Dynamically change Rx HW filter<br>Added 5.13 Stop transmission<br>Added 5.14 Protocol error detection<br>Added 5.15 Dynamically change any baud rate parameters<br>6.7 Hardware events<br>Added new Hardware event to Table 7 Hardware events<br>Added 7.2.8 Can_RxFilterDataType<br>Added 7.2.9 Can_ControllerFilterType<br>Added 7.2.10 Can_BaudrateCustomType<br>7.3.2 Vendor-specific error codes<br>Added new error code to Table 9 Vendor-specific error codes<br>7.3.5 API service IDs<br>Added new API to Table 12 API service IDs<br>Added 7.4.20 Can_SetRxFilter<br>Added 7.4.21 Can_SetHwOperationMode<br>Added 7.4.22 Can_Flush<br>Added 7.4.23 Can_SetBaudrateAnyParameters<br>Added 7.5.5.3 Protocol error callout API<br>8.1 TT_CANFD<br>Added new register access timing to Table 14 TT_CANFD access register table |
| *N | 2024-11-14 | Updated 4.1.27 CanProtocolErrorCalloutFunction<br>Updated 4.2 Can settings configuration<br>Updated 5.11 Dynamically change Rx HW filter<br>Updated 5.12 Bus monitoring mode<br>Updated 5.14 Protocol error detection<br>Updated 5.15 Dynamically change any baud rate parameters<br>Updated 6.7 Hardware events<br>Updated 7.3.2 Vendor-specific error codes<br>Updated 7.4.6 Can_Write<br>Updated 7.4.14 Can_CheckBaudrate<br>Updated 7.4.20 Can_SetRxFilter<br>Updated 7.4.21 Can_SetHwOperationMode<br>Updated 7.4.22 Can_Flush<br>Updated 7.4.23 Can_SetBaudrateAnyParameters<br>Updated 7.5.5.3 Protocol error callout API<br>Updated 8.1 TT_CANFD |

**Edition 2024-11-14**

**Published by**

**Infineon Technologies AG**

**81726 Munich, Germany**

**Do you have a question about this document?**

**Email:**

erratum@infineon.com

**Document reference**

**002-23388 Rev. *N**